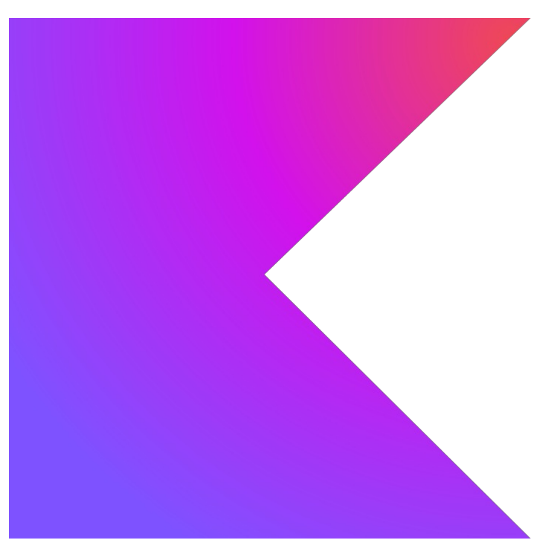


Kotlin Language Documentation 1.9.0

Table of Contents

Kotlin Docs 64

Get started with Kotlin 64

Install Kotlin 64

Create your powerful application with Kotlin 64

Is anything missing? 66

Welcome to our tour of Kotlin! 66

Hello world 66

Variables 67

String templates 67

Practice 68

Next step 68

Basic types 68

Practice 69

Next step 70

Collections 70

List 70

Set 71

Map 72

Practice 74

Next step 75

Control flow 75

Conditional expressions 75

Ranges 76

Loops 77

Practice 78

Next step 80

Functions 80
Named arguments 80
Default parameter values 81
Functions without return 81
Single-expression functions 81
Functions practice 82
Lambda expressions 83
Lambda expressions practice 86
Next step 86
Classes 86
Properties 86
Create instance 87
Access properties 87
Member functions 88
Data classes 88
Practice 90
Next step 90
Null safety 91
Nullable types 91
Check for null values 91
Use safe calls 92
Use Elvis operator 92
Practice 92
What's next? 93
Kotlin Multiplatform 93
Kotlin Multiplatform use cases 93
Code sharing between platforms 94
Get started 94
Kotlin for server side 94

Frameworks for server-side development with Kotlin 95
Deploying Kotlin server-side applications 95
Products that use Kotlin on the server side 95
Next steps 96

Kotlin for Android 96

Kotlin Wasm 96
Browser support 97
Interoperability 97
Compose Multiplatform for Web 97
How to get started 97
Libraries support 97
Feedback 97

Kotlin Native 97
Why Kotlin/Native? 97
Target platforms 98
Interoperability 98
Sharing code between platforms 98
How to get started 98

Kotlin for JavaScript 99
Kotlin/JS IR compiler 99
Kotlin/JS frameworks 99
Join the Kotlin/JS community 100

Kotlin for data science 100
Interactive editors 100
Libraries 102

Kotlin for competitive programming 103
Simple example: Reachable Numbers problem 104

Functional operators example: Long Number problem 105
More tips and tricks 106
Learning Kotlin 107
What's new in Kotlin 1.9.0 107
IDE support 108
New Kotlin K2 compiler updates 108
Language 110
Kotlin/JVM 111
Kotlin/Native 111
Kotlin Multiplatform 114
Kotlin/Wasm 115
Kotlin/JS 116
Gradle 117
Standard library 120
Documentation updates 125
Install Kotlin 1.9.0 125
Compatibility guide for Kotlin 1.9.0 125
What's new in Kotlin 1.8.20 125
IDE support 126
New Kotlin K2 compiler updates 126
Language 127
New Kotlin/Wasm target 131
Kotlin/JVM 132
Kotlin/Native 133
Kotlin Multiplatform 135
Kotlin/JavaScript 138
Gradle 139
Standard library 142
Serialization updates 144

Documentation updates 145
Install Kotlin 1.8.20 145

What's new in Kotlin 1.9.0-RC 145
IDE support 146
New Kotlin K2 compiler updates 146
Stable replacement of the enum class values function 146
Stable ..< operator for open-ended ranges 147
New common function to get regex capture group by name 147
New path utility to create parent directories 147
Preview of Gradle configuration cache in Kotlin Multiplatform 148
Changes for Android target support in Kotlin Multiplatform 148
No object initialization when accessing constant values in Kotlin/Native 148
Ability to configure standalone mode for iOS simulator tests in Kotlin/Native 148
How to update to Kotlin 1.9.0-RC 149

What's new in Kotlin 1.8.0 149
IDE support 149
Kotlin/JVM 149
Kotlin/Native 150
Kotlin Multiplatform: A new Android source set layout 151
Kotlin/JS 154
Gradle 155
Standard library 158
Documentation updates 160
Install Kotlin 1.8.0 161
Compatibility guide for Kotlin 1.8.0 161

What's new in Kotlin 1.7.20 161
Support for Kotlin K2 compiler plugins 162
Language 162
Kotlin/JVM 166

Kotlin/Native 168
Kotlin/JS 169
Gradle 169
Standard library 170
Documentation updates 172
Install Kotlin 1.7.20 172

What's new in Kotlin 1.7.0 173
New Kotlin K2 compiler for the JVM in Alpha 173
Language 174
Kotlin/JVM 176
Kotlin/Native 176
Kotlin/JS 178
Standard library 179
Gradle 183
Migrating to Kotlin 1.7.0 187

What's new in Kotlin 1.6.20 188
Language 188
Kotlin/JVM 190
Kotlin/Native 191
Kotlin Multiplatform 194
Kotlin/JS 196
Security 197
Gradle 199

What's new in Kotlin 1.6.0 201
Language 201
Supporting previous API versions for a longer period 203
Kotlin/JVM 203
Kotlin/Native 204

Kotlin/JS 206
Kotlin Gradle plugin 207
Standard library 207
Tools 210
Coroutines 1.6.0-RC 211
Migrating to Kotlin 1.6.0 211

What's new in Kotlin 1.5.30 211
Language features 212
Kotlin/JVM 215
Kotlin/Native 216
Kotlin Multiplatform 218
Kotlin/JS 220
Gradle 220
Standard library 223
Serialization 1.3.0-RC 226

What's new in Kotlin 1.5.20 226
Kotlin/JVM 226
Kotlin/Native 227
Kotlin/JS 228
Gradle 229
Standard library 229

What's new in Kotlin 1.5.0 230
Language features 230
Kotlin/JVM 232
Kotlin/Native 234
Kotlin/JS 235
Kotlin Multiplatform 235
Standard library 236
kotlin-test library 240

kotlinx libraries 242

Migrating to Kotlin 1.5.0 243

What's new in Kotlin 1.4.30 243

Language features 244

Kotlin/JVM 246

Kotlin/Native 247

Kotlin/JS 247

Gradle project improvements 247

Standard library 247

Serialization updates 249

What's new in Kotlin 1.4.20 250

Kotlin/JVM 250

Kotlin/JS 250

Kotlin/Native 252

Kotlin Multiplatform 254

Standard library 254

Kotlin Android Extensions 255

What's new in Kotlin 1.4.0 255

Language features and improvements 255

New tools in the IDE 259

New compiler 261

Kotlin/JVM 264

Kotlin/JS 265

Kotlin/Native 266

Kotlin Multiplatform 267

Gradle project improvements 270

Standard library 272

Stable JSON serialization 277

Scripting and REPL 277
Migrating to Kotlin 1.4.0 278

What's new in Kotlin 1.3 279
Coroutines release 279
Kotlin/Native 279
Multiplatform projects 279
Contracts 279
Capturing when subject in a variable 280
@JvmStatic and @JvmField in companions of interfaces 281
Nested declarations in annotation classes 281
Parameterless main 281
Functions with big arity 282
Progressive mode 282
Inline classes 282
Unsigned integers 282
@JvmDefault 283
Standard library 283
Tooling 285

What's new in Kotlin 1.2 286
Table of contents 286
Multiplatform projects (experimental) 286
Other language features 286
Standard library 289
JVM backend 291
JavaScript backend 291
Tools 292

What's new in Kotlin 1.1 292
Table of contents 292
JavaScript 292

Coroutines (experimental) 292
Other language features 293
Standard library 297
JVM Backend 300
JavaScript backend 301

Kotlin releases 301
Update to a new release 302
IDE support 302
Release details 302

Kotlin roadmap 308
Key priorities 308
Kotlin roadmap by subsystem 309
What's changed since May 2022 311

Basic syntax 312
Package definition and imports 312
Program entry point 312
Print to the standard output 313
Functions 313
Variables 314
Creating classes and instances 314
Comments 315
String templates 315
Conditional expressions 315
for loop 316
while loop 316
when expression 316
Ranges 317
Collections 318

Nullable values and null checks 318
Type checks and automatic casts 319

Idioms 320

Create DTOs (POJOs/POCOs) 320
Default values for function parameters 320
Filter a list 321
Check the presence of an element in a collection 321
String interpolation 321
Instance checks 321
Read-only list 321
Read-only map 321
Access a map entry 321
Traverse a map or a list of pairs 321
Iterate over a range 322
Lazy property 322
Extension functions 322
Create a singleton 322
Instantiate an abstract class 322
If-not-null shorthand 322
If-not-null-else shorthand 323
Execute a statement if null 323
Get first item of a possibly empty collection 323
Execute if not null 323
Map nullable value if not null 323
Return on when statement 323
try-catch expression 323
if expression 324
Builder-style usage of methods that return Unit 324
Single-expression functions 324

Call multiple methods on an object instance (with) 324
Configure properties of an object (apply) 325
Java 7's try-with-resources 325
Generic function that requires the generic type information 325
Nullable Boolean 325
Swap two variables 325
Mark code as incomplete (TODO) 325
What's next? 325

Coding conventions 326
Configure style in IDE 326
Source code organization 326
Naming rules 328
Formatting 329
Documentation comments 336
Avoid redundant constructs 337
Idiomatic use of language features 337
Coding conventions for libraries 341

Basic types 341

Numbers 341
Integer types 341
Floating-point types 342
Literal constants for numbers 342
Numbers representation on the JVM 343
Explicit number conversions 343
Operations on numbers 344

Unsigned integer types 346
Unsigned arrays and ranges 346
Unsigned integers literals 346

Use cases 347

Booleans 347

Characters 348

Strings 348

String literals 349

String templates 349

Arrays 350

Primitive type arrays 350

Type checks and casts 351

is and !is operators 351

Smart casts 351

"Unsafe" cast operator 352

"Safe" (nullable) cast operator 352

Generics type checks and casts 352

Conditions and loops 352

If expression 352

When expression 353

For loops 354

While loops 355

Break and continue in loops 356

Returns and jumps 356

Break and continue labels 356

Return to labels 356

Exceptions 358

Exception classes 358

Checked exceptions 358

The Nothing type 359

Java interoperability 359

Packages and imports 359

Default imports 360

Imports 360

Visibility of top-level declarations 360

Classes 360

Constructors 361

Creating instances of classes 363

Class members 363

Inheritance 363

Abstract classes 363

Companion objects 364

Inheritance 364

Overriding methods 364

Overriding properties 365

Derived class initialization order 365

Calling the superclass implementation 366

Overriding rules 366

Properties 367

Declaring properties 367

Getters and setters 367

Compile-time constants 368

Late-initialized properties and variables 369

Overriding properties 369

Delegated properties 369

Interfaces 370

Implementing interfaces 370

Properties in interfaces 370

Interfaces Inheritance 370

Resolving overriding conflicts 371

Functional (SAM) interfaces 371

SAM conversions 371

Migration from an interface with constructor function to a functional interface 372

Functional interfaces vs. type aliases 372

Visibility modifiers 373

Packages 373

Class members 374

Modules 374

Extensions 375

Extension functions 375

Extensions are resolved statically 375

Nullable receiver 376

Extension properties 376

Companion object extensions 377

Scope of extensions 377

Declaring extensions as members 377

Note on visibility 378

Data classes 378

Properties declared in the class body 379

Copying 379

Data classes and destructuring declarations 380

Standard data classes 380

Sealed classes and interfaces 380

Location of direct subclasses 380

Sealed classes and when expression 381

Generics: in, out, where 381

Variance 382
Type projections 383
Generic functions 385
Generic constraints 385
Type erasure 385
Underscore operator for type arguments 387

Nested and inner classes 387
Inner classes 388
Anonymous inner classes 388

Enum classes 388
Anonymous classes 388
Implementing interfaces in enum classes 389
Working with enum constants 389

Inline value classes 390
Members 390
Inheritance 391
Representation 391
Inline classes vs type aliases 392
Inline classes and delegation 393

Object expressions and declarations 393
Object expressions 393
Object declarations 395

Delegation 398
Overriding a member of an interface implemented by delegation 398

Delegated properties 399
Standard delegates 400
Delegating to another property 400

Storing properties in a map 401
Local delegated properties 402
Property delegate requirements 402
Translation rules for delegated properties 403
Providing a delegate 404

Type aliases 405

Functions 406
Function usage 406
Function scope 410
Generic functions 410
Tail recursive functions 411

High-order functions and lambdas 411
Higher-order functions 411
Function types 412
Lambda expressions and anonymous functions 414

Inline functions 416
noinline 417
Non-local returns 417
Reified type parameters 418
Inline properties 418
Restrictions for public API inline functions 419

Operator overloading 419
Unary operations 419
Binary operations 421
Infix calls for named functions 423

Type-safe builders 424
How it works 424
Scope control: @DslMarker 426

Full definition of the com.example.html package 426

Using builders with builder type inference 428

Writing your own builders 428

How builder inference works 430

Null safety 431

Nullable types and non-null types 431

Checking for null in conditions 432

Safe calls 433

Nullable receiver 433

Elvis operator 434

The !! operator 434

Safe casts 434

Collections of a nullable type 434

What's next? 434

Equality 435

Structural equality 435

Referential equality 435

Floating-point numbers equality 435

This expressions 435

Qualified this 435

Implicit this 436

Asynchronous programming techniques 436

Threading 436

Callbacks 437

Futures, promises, and others 437

Reactive extensions 438

Coroutines 438

Coroutines 439

How to start 439

Sample projects 439

Annotations 439

Usage 440

Constructors 440

Instantiation 441

Lambdas 441

Annotation use-site targets 441

Java annotations 442

Repeatable annotations 443

Destructuring declarations 444

Example: returning two values from a function 444

Example: destructuring declarations and maps 445

Underscore for unused variables 445

Destructuring in lambdas 445

Reflection 446

JVM dependency 446

Class references 446

Callable references 447

Get started with Kotlin Multiplatform for mobile 450

Next step 451

Join the community 451

Set up an environment 451

Install the necessary tools 451

Check your environment 452

Possible issues and solutions 452

Next step 454

Get help 454

Create your first cross-platform app 454

Create the project from a template 454

Examine the project structure 456

Run your application 458

Update your application 463

Next step 467

Get help 467

Add dependencies to your project 467

Dependency types 467

Add a multiplatform dependency 468

Next step 469

Get help 470

Upgrade your app 470

Add more dependencies 470

Create API requests 471

Update Android and iOS apps 472

Next step 475

Get help 476

Wrap up your project 476

Understand mobile project structure 476

Root project 477

Shared module 478

Android application 482

iOS application 483

Make your Android application work on iOS – tutorial 486

Prepare an environment for development 486

Make your code cross-platform 486
Make your cross-platform application work on iOS 494
Enjoy the results – update the logic only once 503
What else to share? 506
What's next? 506

Publish your application 507
Android app 507
iOS app 507

Create a multiplatform app using Ktor and SQLDelight – tutorial 507
Before you start 508
Create a Multiplatform project 509
Add dependencies to the multiplatform library 512
Create an application data model 513
Configure SQLDelight and implement cache logic 513
Implement an API service 516
Build an SDK 517
Create the Android application 518
Create the iOS application 523
What's next? 527

Get started with Kotlin Multiplatform 527
Start from scratch 527
Dive deep into Kotlin Multiplatform 527
Get help 527

Understand Multiplatform project structure 528
Multiplatform plugin 528
Targets 528
Source sets 529
Compilations 530

Set up targets for Kotlin Multiplatform 531
Distinguish several targets for one platform 531

Build a full-stack web app with Kotlin Multiplatform 532
Create the project 532
Build the backend 534
Set up the frontend 538
Build the frontend 541
Include a database to store data 545
Deploy to the cloud 548
What's next 549

Create and publish a multiplatform library – tutorial 549
Set up the environment 550
Create a project 550
Write cross-platform code 552
Provide platform-specific implementations 552
Test your library 554
Publish your library to the local Maven repository 555
Publish your library to the external Maven Central repository 556
Add a dependency on the published library 558
What's next? 558

Publishing multiplatform libraries 558
Structure of publications 559
Avoid duplicate publications 559
Publish an Android library 560

Share code on platforms 561
Share code on all platforms 561
Share code on similar platforms 561
Share code in libraries 562

What's next? 562

Connect to platform-specific APIs 563

Examples 563

Rules for expected and actual declarations 567

Hierarchical project structure 568

Default hierarchy 568

Target shortcuts 571

Manual configuration 573

Android source set layout 575

Check the compatibility 575

Rename Kotlin source sets 575

Move source files 575

Move the AndroidManifest.xml file 576

Check the relation between Android and common tests 576

Adjust the implementation of Android flavors 577

Adding dependencies on multiplatform libraries 577

Dependency on a Kotlin library 577

Dependency on Kotlin Multiplatform libraries 579

Dependency on another multiplatform project 580

What's next? 581

Adding Android dependencies 581

What's next? 582

Adding iOS dependencies 582

With CocoaPods 582

Without CocoaPods 583

What's next? 585

Run tests with Kotlin Multiplatform 585

Required dependencies 586
Run tests for one or more targets 586
Test shared code 586
Configure compilations 586
Configure all compilations 587
Configure compilations for one target 587
Configure one compilation 588
Create a custom compilation 588
Use Java sources in JVM compilations 589
Configure interop with native languages 590
Compilation for Android 591
Compilation of the source set hierarchy 592
Build final native binaries (Experimental DSL) 592
Declare binaries 593
Configure binaries 594
Build final native binaries 597
Declare binaries 598
Access binaries 599
Export dependencies to binaries 600
Build universal frameworks 602
Build XCFrameworks 602
Customize the Info.plist file 603
Multiplatform Gradle DSL reference 604
Id and version 604
Top-level blocks 604
Targets 605
Source sets 611
Compilations 613

Dependencies 616
Language settings 617
Kotlin Multiplatform for mobile samples 618
Kotlin Multiplatform for mobile FAQ 620
What is Kotlin Multiplatform for mobile? 620
Can I share UIs with Kotlin Multiplatform? 620
What is the Kotlin Multiplatform Mobile plugin? 620
What is Kotlin/Native and how does it relate to Kotlin Multiplatform? 621
What are the plans for the technology evolution? 621
Can I run an iOS application on Microsoft Windows or Linux? 622
Where can I get complete examples to play with? 622
In which IDE should I work on my cross-platform app? 622
How can I write concurrent code in Kotlin Multiplatform projects? 622
How can I speed up my Kotlin Multiplatform module compilation for iOS? 622
What platforms do you support? 622
Introduce cross-platform mobile development to your team 623
Start with empathy 623
Explain how it works 623
Show the value 623
Offer proof 624
Prepare for questions 624
Be supportive 625
Compatibility guide for Kotlin Multiplatform 625
New approach to auto-generated targets 625
Changes in Gradle input and output compile tasks 626
New configuration names for dependencies on the compilation 626
Deprecated Gradle properties for hierarchical structure support 627
Deprecated support of multiplatform libraries published in the legacy mode 628

Deprecated API for adding Kotlin source sets directly to the Kotlin compilation 628
Migration from kotlin-js Gradle plugin to kotlin-multiplatform Gradle plugin 629
Rename of android target to androidTarget 631

Kotlin Multiplatform Mobile plugin releases 631
Update to the new release 631
Release details 631

Get started with Kotlin/JVM 636
Create a project 636
Create an application 637
Run the application 638
What's next? 639

Comparison to Java 639
Some Java issues addressed in Kotlin 639
What Java has that Kotlin does not 640
What Kotlin has that Java does not 640
What's next? 640

Calling Java from Kotlin 640
Getters and setters 641
Java synthetic property references 641
Methods returning void 642
Escaping for Java identifiers that are keywords in Kotlin 642
Null-safety and platform types 642
Mapped types 647
Java generics in Kotlin 649
Java arrays 650
Java varargs 650
Operators 651
Checked exceptions 651

Object methods ... 651
Inheritance from Java classes ... 652
Accessing static members ... 652
Java reflection ... 652
SAM conversions ... 652
Using JNI with Kotlin ... 652
Using Lombok-generated declarations in Kotlin ... 653

Calling Kotlin from Java ... 653
Properties ... 653
Package-level functions ... 653
Instance fields ... 654
Static fields ... 655
Static methods ... 656
Default methods in interfaces ... 656
Visibility ... 658
KClass ... 658
Handling signature clashes with @JvmName ... 658
Overloads generation ... 659
Checked exceptions ... 659
Null-safety ... 660
Variant generics ... 660

Get started with Spring Boot and Kotlin ... 661
Next step ... 661
Join the community ... 661

Create a Spring Boot project with Kotlin ... 662
Before you start ... 662
Create a Spring Boot project ... 662
Explore the project Gradle build file ... 665
Explore the generated Spring Boot application ... 666

Create a controller 667
Run the application 668
Next step 669

Add a data class to Spring Boot project 669
Update your application 669
Run the application 671
Next step 671

Add database support for Spring Boot project 672
Add database support 672
Update the MessageController class 673
Update the MessageService class 673
Configure the database 674
Add messages to database via HTTP request 674
Retrieve messages by id 676
Run the application 677
Next step 678

Use Spring Data CrudRepository for database access 678
Update your application 679
Run the application 680
Next step 680

Test code using JUnit in JVM – tutorial 680
Add dependencies 680
Add the code to test it 681
Create a test 681
Run a test 682
What's next 683

Mixing Java and Kotlin in one project – tutorial 683
Adding Java source code to an existing Kotlin project 683

Adding Kotlin source code to an existing Java project 684
Converting an existing Java file to Kotlin with J2K 685

Using Java records in Kotlin 686
Using Java records from Kotlin code 687
Declare records in Kotlin 687
Further discussion 687

Strings in Java and Kotlin 687
Concatenate strings 688
Build a string 688
Create a string from collection items 688
Set default value if the string is blank 689
Replace characters at the beginning and end of a string 689
Replace occurrences 690
Split a string 690
Take a substring 690
Use multiline strings 691
What's next? 692

Collections in Java and Kotlin 692
Operations that are the same in Java and Kotlin 692
Operations that differ a bit 694
Operations that don't exist in Java's standard library 695
Mutability 696
Covariance 697
Ranges and progressions 697
Comparison by several criteria 698
Sequences 699
Removal of elements from a list 699
Traverse a map 700
Get the first and the last items of a possibly empty collection 700

Create a set from a list 700
Group elements 701
Filter elements 701
Collection transformation operations 702
What's next? 703

Nullability in Java and Kotlin 703
Support for nullable types 704
Platform types 705
Checking the result of a function call 705
Default values instead of null 706
Functions returning a value or null 706
Aggregate operations 707
Casting types safely 707
What's next? 708

Introduction 708
Cognitive complexity 708
What's next? 709

Readability 709
API consistency 709
Use a builder DSL 709
Use constructor-like functions where applicable 711
Use member and extension functions appropriately 711
Avoid using Boolean arguments in functions 712
What's next? 712

Predictability 712
Use sealed interfaces 713
Hide implementations with sealed classes 713
Validate your inputs and state 713

Avoid arrays in public signatures 714
Avoid varargs 715
What's next? 715

Debuggability 715
Always provide a toString() method 715
What's next? 717

Backward compatibility 717
Definition of backward compatibility 718
"Don't do" recommendations 718
The @PublishedApi annotation 721
The @RequiresOptIn annotation 721
Explicit API mode 722
Tools designed to enforce backward compatibility 722

Get started with Kotlin/Native in IntelliJ IDEA 723
Create a new Kotlin/Native project in IntelliJ IDEA 723
Build and run the application 725
Update the application 726
What's next? 728

Get started with Kotlin/Native using Gradle 728
Create project files 728
Build and run the application 729
Open the project in an IDE 729
What's next? 730

Get started with Kotlin/Native using the command-line compiler 730
Obtain the compiler 730
Write "Hello Kotlin/Native" program 730
Compile the code from the console 730

Interoperability with C 730

Platform libraries 731
Simple example 731
Create bindings for a new library 731
Bindings 733
Mapping primitive data types from C – tutorial 737
Types in C language 737
Example C library 738
Inspect generated Kotlin APIs for a C library 738
Primitive types in kotlin 740
Fix the code 740
Next steps 741
Mapping struct and union types from C – tutorial 741
Mapping struct and union C types 741
Inspect Generated Kotlin APIs for a C library 742
Struct and union types in Kotlin 743
Use struct and union types from Kotlin 743
Run the code 745
Next steps 746
Mapping function pointers from C – tutorial 746
Mapping function pointer types from C 746
Inspect generated Kotlin APIs for a C library 746
C function pointers in Kotlin 748
Pass Kotlin function as C function pointer 748
Use the C function pointer from Kotlin 748
Fix the code 748
Next Steps 749
Mapping Strings from C – tutorial 749
Working with C strings 749

Inspect generated Kotlin APIs for a C library 750
Strings in Kotlin 751
Pass Kotlin string to C 751
Read C Strings in Kotlin 752
Receive C string bytes from Kotlin 752
Fix the Code 752
Next steps 753

Create an app using C Interop and libcurl – tutorial 753
Create a Kotlin/Native project 753
Create a definition file 755
Add interoperability to the build process 756
Write the application code 757
Compile and run the application 757

Interoperability with Swift/Objective-C 758
Usage 758
Mappings 759
Casting between mapped types 764
Subclassing 764
C features 765
Export of KDoc comments to generated Objective-C headers 765
Unsupported 766

Kotlin/Native as an Apple framework – tutorial 766
Create a Kotlin library 766
Generated framework headers 768
Garbage collection and reference counting 771
Use the code from Objective-C 771
Use the code from Swift 772
Xcode and framework dependencies 772
Next steps 772

CocoaPods overview and setup 773
Set up an environment to work with CocoaPods 773
Add and configure Kotlin CocoaPods Gradle plugin 774
Update Podfile for Xcode 775
Possible issues and solutions 775
Add dependencies on a Pod library 776
From the CocoaPods repository 777
On a locally stored library 777
From a custom Git repository 778
From a custom Podspec repository 779
With custom cinterop options 779
Use a Kotlin Gradle project as a CocoaPods dependency 780
Xcode project with one target 780
Xcode project with several targets 781
CocoaPods Gradle plugin DSL reference 782
Enable the plugin 782
cocoapods block 782
pod() function 784
Kotlin/Native libraries 785
Kotlin compiler specifics 785
cinterop tool specifics 785
klib utility 786
Several examples 786
Advanced topics 787
Platform libraries 788
POSIX bindings 788
Popular native libraries 788

Availability by default 788

Kotlin/Native as a dynamic library – tutorial 788

Create a Kotlin library 789

Generated headers file 791

Use generated headers from C 794

Compile and run the example on Linux and macOS 794

Compile and run the example on Windows 794

Next steps 795

Kotlin/Native memory management 795

Garbage collector 795

Memory consumption 796

Unit tests in the background 796

Legacy memory manager 797

What's next 797

iOS integration 797

Threads 797

Garbage collection and lifecycle 798

Support for background state and App Extensions 800

Migrate to the new memory manager 801

Update Kotlin 801

Update dependencies 801

Update your code 801

Support both new and legacy memory managers 802

What's next 802

Immutability and concurrency in Kotlin/Native 803

Concurrency in Kotlin/Native 803

Concurrency overview 805

Rules for state sharing 806

Immutable and frozen state 806
Global state 807
Current and future models 808
Concurrent mutability 808
Atomics 809
Thread-isolated state 810
Low-level capabilities 811
Concurrency and coroutines 811
Coroutines 812
Multithreaded coroutines 813
Alternatives to kotlinx-coroutines 814
Debugging Kotlin/Native 815
Produce binaries with debug info with Kotlin/Native compiler 815
Breakpoints 816
Stepping 817
Variable inspection 817
Known issues 818
Symbolicating iOS crash reports 818
Producing .dSYM for release Kotlin binaries 818
Make frameworks static when using rebuild from bitcode 819
Decode inlined stack frames 819
Kotlin/Native target support 819
Tier 1 820
Tier 2 820
Tier 3 821
Deprecated targets 821
For library authors 822

Tips for improving Kotlin/Native compilation times 822
General recommendations 822
Gradle configuration 822
Windows OS configuration 823
License files for the Kotlin/Native binaries 823
Kotlin/Native FAQ 824
How do I run my program? 824
What is Kotlin/Native memory management model? 824
How do I create a shared library? 824
How do I create a static library or an object file? 825
How do I run Kotlin/Native behind a corporate proxy? 825
How do I specify a custom Objective-C prefix/name for my Kotlin framework? 825
How do I rename the iOS framework? 825
How do I enable bitcode for my Kotlin framework? 826
Why do I see InvalidMutabilityException? 826
How do I make a singleton object mutable? 826
How can I compile my project with unreleased versions of Kotlin/Native? 826
Get started with Kotlin/Wasm in IntelliJ IDEA 826
Enable an experimental Kotlin/Wasm Wizard in IntelliJ IDEA 827
Create a new Kotlin/Wasm project 827
Build and run the application 828
Update your application 831
What's next? 831
Add dependencies on Kotlin libraries to Kotlin/Wasm project 831
Supported Kotlin libraries for Kotlin/Wasm 832
Enable libraries in your project 833
What's next? 833
Interoperability with JavaScript 833

Use JavaScript code from Kotlin 833
Use Kotlin code from JavaScript 835
Kotlin types in JavaScript 836
Exception handling 836
Workarounds for Kotlin/JS features non-supported in Kotlin/Wasm 837

Set up a Kotlin/JS project 837
Execution environments 838
Dependencies 839
run task 840
test task 841
webpack bundling 842
CSS 843
Node.js 844
Yarn 845
Distribution target directory 847
Module name 848
package.json customization 848
Troubleshooting 848

Run Kotlin/JS 848
Run the Node.js target 849
Run the browser target 850

Development server and continuous compilation 851

Debug Kotlin/JS code 853
Debug in browser 853
Debug in the IDE 854
Debug in Node.js 857
What's next? 857
If you run into any problems 857

Run tests in Kotlin/JS 857

Kotlin/JS dead code elimination 861

Exclude declarations from DCE 861

Disable DCE 862

Kotlin/JS IR compiler 862

Lazy initialization of top-level properties 863

Incremental compilation for development binaries 863

Output .js files: one per module or one for the whole project 863

Ignoring compilation errors 863

Minification of member names in production 864

Preview: generation of TypeScript declaration files (d.ts) 864

Current limitations of the IR compiler 864

Migrating existing projects to the IR compiler 865

Authoring libraries for the IR compiler with backwards compatibility 865

Migrating Kotlin/JS projects to the IR compiler 865

Convert JS- and React-related classes and interfaces to external interfaces 865

Convert properties of external interfaces to var 866

Convert functions with receivers in external interfaces to regular functions 866

Create plain JS objects for interoperability 866

Replace toString() calls on function references with .name 867

Explicitly specify binaries.executable() in the build script 867

Additional troubleshooting tips when working with the Kotlin/JS IR compiler 867

Browser and DOM API 868

Interaction with the DOM 868

Use JavaScript code from Kotlin 869

Inline JavaScript 869

external modifier 869

Dynamic type 871

Use dependencies from npm 872

Use Kotlin code from JavaScript 873

Isolating declarations in a separate JavaScript object in plain mode 873

Package structure 873

Kotlin types in JavaScript 875

JavaScript modules 875

Browser targets 875

JavaScript libraries and Node.js files 876

@JsModule annotation 876

Kotlin/JS reflection 878

Class references 878

KType and typeOf() 878

Example 878

Typesafe HTML DSL 879

Build a web application with React and Kotlin/JS — tutorial 880

Before you start 880

Create a web app draft 883

Design app components 888

Compose components 892

Add more components 893

Use packages from npm 895

Use an external REST API 898

Deploy to production and the cloud 901

What's next 902

Get started with Kotlin custom scripting – tutorial 903

Project structure 903

Before you start 903

Create a project 903
Add scripting modules 904
Create a script definition 906
Create a scripting host 907
Run scripts 908
What's next? 909

Collections overview 909
Collection types 910

Constructing collections 914
Construct from elements 914
Create with collection builder functions 914
Empty collections 914
Initializer functions for lists 915
Concrete type constructors 915
Copy 915
Invoke functions on other collections 916

Iterators 916
List iterators 917
Mutable iterators 917

Ranges and progressions 918
Progression 919

Sequences 920
Construct 920
Sequence operations 921
Sequence processing example 921

Collection operations overview 923
Extension and member functions 923
Common operations 923

Write operations 924

Collection transformation operations 925

Map 925

Zip 925

Associate 926

Flatten 927

String representation 927

Filtering collections 928

Filter by predicate 928

Partition 929

Test predicates 929

Plus and minus operators 930

Grouping 930

Retrieve collection parts 931

Slice 931

Take and drop 931

Chunked 932

Windowed 932

Retrieve single elements 933

Retrieve by position 933

Retrieve by condition 934

Retrieve with selector 934

Random element 935

Check element existence 935

Ordering 935

Natural order 936

Custom orders 937

Reverse order 937
Random order 938

Aggregate operations 938
Fold and reduce 939

Collection write operations 940
Adding elements 940
Removing elements 941
Updating elements 942

List-specific operations 942
Retrieve elements by index 942
Retrieve list parts 942
Find element positions 942
List write operations 944

Set-specific operations 945

Map-specific operations 946
Retrieve keys and values 946
Filter 947
Plus and minus operators 947
Map write operations 947

Scope functions 949
Function selection 950
Distinctions 950
Functions 953
takeIf and takeUnless 956

Opt-in requirements 957
Opt in to using API 957
Require opt-in for API 960

Opt-in requirements for pre-stable APIs 960

Coroutines guide 961

Table of contents 961

Additional references 961

Coroutines basics 961

Your first coroutine 962

Extract function refactoring 962

Scope builder 963

Scope builder and concurrency 963

An explicit job 964

Coroutines are light-weight 964

Coroutines and channels – tutorial 965

Before you start 965

Blocking requests 967

Callbacks 970

Suspending functions 974

Coroutines 975

Concurrency 977

Structured concurrency 980

Showing progress 983

Channels 985

Testing coroutines 988

What's next 991

Cancellation and timeouts 991

Cancelling coroutine execution 991

Cancellation is cooperative 992

Making computation code cancellable 993

Closing resources with finally 993

Run non-cancellable block ... 994
Timeout ... 994
Asynchronous timeout and resources ... 995

Composing suspending functions ... 996
Sequential by default ... 997
Concurrent using async ... 997
Lazily started async ... 998
Async-style functions ... 999
Structured concurrency with async ... 1000

Coroutine context and dispatchers ... 1001
Dispatchers and threads ... 1001
Unconfined vs confined dispatcher ... 1002
Debugging coroutines and threads ... 1003
Jumping between threads ... 1004
Job in the context ... 1005
Children of a coroutine ... 1005
Parental responsibilities ... 1006
Naming coroutines for debugging ... 1006
Combining context elements ... 1007
Coroutine scope ... 1007

Asynchronous Flow ... 1009
Representing multiple values ... 1009
Flows are cold ... 1011
Flow cancellation basics ... 1012
Flow builders ... 1012
Intermediate flow operators ... 1013
Terminal flow operators ... 1014
Flows are sequential ... 1015
Flow context ... 1016

Buffering 1017
Composing multiple flows 1020
Flattening flows 1021
Flow exceptions 1023
Exception transparency 1024
Flow completion 1026
Imperative versus declarative 1028
Launching flow 1028
Flow and Reactive Streams 1031

Channels 1031
Channel basics 1031
Closing and iteration over channels 1031
Building channel producers 1032
Pipelines 1032
Prime numbers with pipeline 1033
Fan-out 1034
Fan-in 1035
Buffered channels 1036
Channels are fair 1036
Ticker channels 1037

Coroutine exceptions handling 1038
Exception propagation 1038
CoroutineExceptionHandler 1039
Cancellation and exceptions 1039
Exceptions aggregation 1041
Supervision 1042

Shared mutable state and concurrency 1044
The problem 1044

Volatiles are of no help 1045
Thread-safe data structures 1045
Thread confinement fine-grained 1046
Thread confinement coarse-grained 1047
Mutual exclusion 1047
Select expression (experimental) 1048
Selecting from channels 1048
Selecting on close 1049
Selecting to send 1051
Selecting deferred values 1052
Switch over a channel of deferred values 1052
Debug coroutines using IntelliJ IDEA – tutorial 1054
Create coroutines 1054
Debug coroutines 1055
Debug Kotlin Flow using IntelliJ IDEA – tutorial 1058
Create a Kotlin flow 1058
Debug the coroutine 1059
Add a concurrently running coroutine 1062
Debug a Kotlin flow with two coroutines 1062
Serialization 1063
Libraries 1063
Formats 1064
Example: JSON serialization 1064
Lincheck guide 1065
Add Lincheck to your project 1065
Explore Lincheck 1066
Additional references 1066
Write your first test with Lincheck 1066

Create a project ... 1066
Add required dependencies ... 1066
Write a concurrent counter and run the test ... 1067
Trace the invalid execution ... 1068
Test the Java standard library ... 1069
Next step ... 1070
See also ... 1070

Stress testing and model checking ... 1070
Stress testing ... 1070
Model checking ... 1071
Which testing strategy is better? ... 1072
Configure the testing strategy ... 1072
Scenario minimization ... 1073
Logging data structure states ... 1073
Next step ... 1074

Operation arguments ... 1074
Next step ... 1076

Data structure constraints ... 1076
Next step ... 1078

Progress guarantees ... 1078
Next step ... 1079

Sequential specification ... 1079

Keywords and operators ... 1080
Hard keywords ... 1080
Soft keywords ... 1082
Modifier keywords ... 1082
Special identifiers ... 1083

Operators and special symbols 1083

Gradle 1084

What's next? 1085

Get started with Gradle and Kotlin/JVM 1085

Create a project 1085

Explore the build script 1086

Run the application 1087

What's next? 1088

Configure a Gradle project 1088

Apply the plugin 1088

Targeting the JVM 1089

Targeting multiple platforms 1095

Targeting Android 1095

Targeting JavaScript 1095

Triggering configuration actions with the KotlinBasePlugin interface 1096

Configure dependencies 1096

What's next? 1102

Compiler options in the Kotlin Gradle plugin 1102

How to define options 1102

What's next? 1107

Compilation and caches in the Kotlin Gradle plugin 1107

Incremental compilation 1107

Gradle build cache support 1109

Gradle configuration cache support 1109

The Kotlin daemon and how to use it with Gradle 1109

Defining Kotlin compiler execution strategy 1111

Kotlin compiler fallback strategy 1113

Build reports 1113

What's next? 1115

Support for Gradle plugin variants 1115

Troubleshooting 1115

What's next? 1117

Maven 1117

Plugin and versions 1117

Dependencies 1117

Compile Kotlin-only source code 1118

Compile Kotlin and Java sources 1118

Incremental compilation 1119

Annotation processing 1120

Jar file 1120

Self-contained Jar file 1120

Specifying compiler options 1120

Using BOM 1122

Generating documentation 1122

OSGi 1122

Ant 1122

Getting the Ant tasks 1122

Targeting JVM with Kotlin-only source 1123

Targeting JVM with Kotlin-only source and multiple roots 1123

Targeting JVM with Kotlin and Java source 1123

Targeting JavaScript with single source folder 1123

Targeting JavaScript with Prefix, PostFix and sourcemap options 1124

Targeting JavaScript with single source folder and metaInfo option 1124

References 1124

Introduction 1125

Community 1126

Get started with Dokka 1126

Gradle 1127

Apply Dokka 1127

Generate documentation 1128

Build javadoc.jar 1131

Configuration examples 1132

Configuration options 1136

Maven 1146

Apply Dokka 1147

Generate documentation 1147

Build javadoc.jar 1148

Configuration example 1148

Configuration options 1149

CLI 1154

Get started 1154

Generate documentation 1154

Command line options 1156

JSON configuration 1159

HTML 1166

Generate HTML documentation 1167

Configuration 1167

Customization 1169

Markdown 1171

GFM 1171

Jekyll 1172

Javadoc 1173

Generate Javadoc documentation 1174

Dokka plugins 1175

Apply Dokka plugins 1176
Configure Dokka plugins 1177
Notable plugins 1178
Module documentation 1179
File format 1179
Pass files to Dokka 1180
IDEs for Kotlin development 1180
IntelliJ IDEA 1180
Android Studio 1180
Eclipse 1180
Compatibility with the Kotlin language versions 1181
Other IDEs support 1181
What's next? 1181
Migrate to Kotlin code style 1181
Kotlin coding conventions and IntelliJ IDEA formatter 1181
Differences between "Kotlin coding conventions" and "IntelliJ IDEA default code style" 1181
Migration to a new code style discussion 1182
Migration to a new code style 1182
Store old code style in project 1183
Run code snippets 1183
IDE: scratches and worksheets 1184
Browser: Kotlin Playground 1185
Command line: ki shell 1187
Kotlin and continuous integration with TeamCity 1189
Gradle, Maven, and Ant 1190
IntelliJ IDEA Build System 1190
Other CI servers 1192
Document Kotlin code: KDoc 1192

KDoc syntax 1192
Inline markup 1193
What's next? 1194
Kotlin and OSGi 1194
Maven 1194
Gradle 1194
FAQ 1195
Kotlin command-line compiler 1195
Install the compiler 1195
Create and run an application 1196
Compile a library 1196
Run the REPL 1196
Run scripts 1197
Kotlin compiler options 1197
Compiler options 1197
Common options 1198
Kotlin/JVM compiler options 1199
Kotlin/JS compiler options 1200
Kotlin/Native compiler options 1201
All-open compiler plugin 1203
Gradle 1204
Maven 1204
Spring support 1204
Command-line compiler 1205
No-arg compiler plugin 1205
Gradle 1205
Maven 1206
JPA support 1206

Command-line compiler 1207

SAM-with-receiver compiler plugin 1207

Gradle 1207

Maven 1207

Command-line compiler 1208

kapt compiler plugin 1208

Using in Gradle 1208

Annotation processor arguments 1209

Gradle build cache support 1209

Improving the speed of builds that use kapt 1209

Compile avoidance for kapt 1211

Incremental annotation processing 1211

Java compiler options 1211

Non-existent type correction 1211

Using in Maven 1212

Using in IntelliJ build system 1212

Using in CLI 1212

Generating Kotlin sources 1213

AP/Javac options encoding 1213

Keeping Java compiler's annotation processors 1213

Lombok compiler plugin 1214

Supported annotations 1214

Gradle 1214

Maven 1215

Using with kapt 1215

Command-line compiler 1216

Kotlin Symbol Processing API 1216

Overview 1216

How KSP looks at source files 1217
SymbolProcessorProvider: the entry point 1218
Resources 1218
Supported libraries 1219

KSP quickstart 1220
Create a processor of your own 1220
Use your own processor in a project 1221
Pass options to processors 1222
Make IDE aware of generated code 1223

Why KSP 1224
KSP makes creating lightweight compiler plugins easier 1224
Comparison to kotlinc compiler plugins 1224
Comparison to reflection 1224
Comparison to kapt 1225
Limitations 1225

KSP examples 1225
Get all member functions 1225
Check whether a class or function is local 1225
Find the actual class or interface declaration that the type alias points to 1225
Collect suppressed names in a file annotation 1226

How KSP models Kotlin code 1226
Type and resolution 1226

Java annotation processing to KSP reference 1227
Program elements 1227
Types 1227
Misc 1228
Details 1229

Incremental processing 1236

Aggregating vs Isolating 1236
Example 1 1237
Example 2 1237
How file dirtiness is determined 1238
Reporting bugs 1238
Multiple round processing 1238
Changes to your processor 1238
Multiple round behavior 1239
Advanced 1240
KSP with Kotlin Multiplatform 1240
Compilation and processing 1240
Avoid the ksp(...) configuration on KSP 1.0.1+ 1240
Running KSP from command line 1241
KSP FAQ 1241
Why KSP? 1242
Why is KSP faster than kapt? 1242
Is KSP Kotlin-specific? 1242
How to upgrade KSP? 1242
Can I use a newer KSP implementation with an older Kotlin compiler? 1242
How often do you update KSP? 1242
Besides Kotlin, are there other version requirements to libraries? 1243
What is KSP's future roadmap? 1243
Learning materials overview 1243
Kotlin Koans 1243
Kotlin tips 1244
null + null in Kotlin 1244
Deduplicating collection items 1244

The suspend and inline mystery 1245
Unshadowing declarations with their fully qualified name 1245
Return and throw with the Elvis operator 1245
Destructuring declarations 1246
Operator functions with nullable values 1246
Timing code 1247
Improving loops 1247
Strings 1248
Doing more with the Elvis operator 1248
Kotlin collections 1249
What's next? 1249

Kotlin books 1249

Advent of Code puzzles in idiomatic Kotlin 1252
Advent of Code 2021 1253
Advent of Code 2020 1255
What's next? 1260

Learning Kotlin with JetBrains Academy plugin 1260

Teaching Kotlin with JetBrains Academy plugin 1260

Participate in the Kotlin Early Access Preview 1260
How the EAP can help you be more productive with Kotlin 1261
Build details 1261

Install the EAP Plugin for IntelliJ IDEA or Android Studio 1261
If you run into any problems 1263

Configure your build for EAP 1263
Configure in Gradle 1263
Configure in Maven 1264

FAQ 1265

What is Kotlin? 1265
What is the current version of Kotlin? 1265
Is Kotlin free? 1265
Is Kotlin an object-oriented language or a functional one? 1265
What advantages does Kotlin give me over the Java programming language? 1265
Is Kotlin compatible with the Java programming language? 1265
What can I use Kotlin for? 1266
Can I use Kotlin for Android development? 1266
Can I use Kotlin for server-side development? 1266
Can I use Kotlin for web development? 1266
Can I use Kotlin for desktop development? 1266
Can I use Kotlin for native development? 1266
What IDEs support Kotlin? 1266
What build tools support Kotlin? 1266
What does Kotlin compile down to? 1266
Which versions of JVM does Kotlin target? 1267
Is Kotlin hard? 1267
What companies are using Kotlin? 1267
Who develops Kotlin? 1267
Where can I learn more about Kotlin? 1267
Are there any books on Kotlin? 1267
Are any online courses available for Kotlin? 1267
Does Kotlin have a community? 1267
Are there Kotlin events? 1267
Is there a Kotlin conference? 1268
Is Kotlin on social media? 1268
Any other online Kotlin resources? 1268
Where can I get an HD Kotlin logo? 1268

Kotlin Evolution 1268

Principles of Pragmatic Evolution 1268
Incompatible changes 1269
Decision making 1269
Feature releases and incremental releases 1269
Libraries 1270
Compiler keys 1270
Compatibility tools 1271
Stability of Kotlin components 1271
Stability levels explained 1271
GitHub badges for Kotlin components 1272
Stability of subcomponents 1272
Current stability of Kotlin components 1272
Stability of Kotlin components (pre 1.4) 1274
Compatibility guide for Kotlin 1.9 1275
Basic terms 1275
Language 1276
Standard library 1283
Tools 1284
Compatibility guide for Kotlin 1.8 1285
Basic terms 1286
Language 1286
Standard library 1294
Tools 1295
Compatibility guide for Kotlin 1.7.20 1297
Basic terms 1297
Language 1298
Compatibility guide for Kotlin 1.7 1298
Basic terms 1298

Language 1299
Standard library 1303
Tools 1304
Compatibility guide for Kotlin 1.6 1308
Basic terms 1308
Language 1308
Standard library 1313
Tools 1316
Compatibility guide for Kotlin 1.5 1318
Basic terms 1318
Language and stdlib 1319
Tools 1326
Compatibility guide for Kotlin 1.4 1327
Basic terms 1327
Language and stdlib 1327
Tools 1341
Compatibility guide for Kotlin 1.3 1342
Basic terms 1342
Incompatible changes 1342
Compatibility modes 1350
What is cross-platform mobile development? 1350
Cross-platform mobile development: definition and solutions 1351
Is cross-platform mobile development right for you? 1352
The most popular cross-platform solutions 1354
Conclusion 1355
Native and cross-platform app development: how to choose? 1355
What is native mobile app development? 1355

What is cross-platform app development? 1355
Six key aspects to help you choose between cross-platform app development and the native 1357
When should you choose cross-platform app development? 1359
When should you choose native app development? 1359
The Six Most Popular Cross-Platform App Development Frameworks 1359
What is a cross-platform app development framework? 1360
Popular cross-platform app development frameworks 1360
How do you choose the right cross-platform app development framework for your project? 1362
Key takeaways 1363
Google Summer of Code with Kotlin 1363
Kotlin contributor guidelines for Google Summer of Code (GSoC) 1364
Project ideas 1364
Security 1367
Kotlin documentation as PDF 1368
Contribution 1368
Participate in Early Access Preview 1368
Contribute to the compiler and standard library 1368
Contribute to the Kotlin IDE plugin 1368
Contribute to other Kotlin libraries and tools 1368
Contribute to the documentation 1368
Create tutorials or videos 1369
Translate documentation to other languages 1369
Hold events and presentations 1369
KUG guidelines 1369
How to run a KUG? 1369
Support for KUGs from JetBrains 1370
Support from JetBrains for other tech communities 1370

Kotlin Night guidelines 1370

Event guidelines 1370

Event requirements 1370

JetBrains support 1371

Kotlin brand assets 1371

Kotlin Logo 1371

Kotlin mascot 1372

Kotlin User Group brand assets 1373

Kotlin Night brand assets 1376

Kotlin Docs

Get started with Kotlin

Kotlin is a modern but already mature programming language designed to make developers happier. It's concise, safe, interoperable with Java and other languages, and provides many ways to reuse code between multiple platforms for productive programming.

To start, why not take our tour of Kotlin? This tour covers the fundamentals of the Kotlin programming language.

Start the Kotlin tour

Install Kotlin

Kotlin is included in each IntelliJ IDEA and Android Studio release. Download and install one of these IDEs to start using Kotlin.

Create your powerful application with Kotlin

Backend app

Here is how you can take the first steps in developing Kotlin server-side applications.

1. Create your first backend application:
   - To start from scratch, create a basic JVM application with the IntelliJ IDEA project wizard.
   - If you prefer more robust examples, choose one of the frameworks below and create a project:

Spring	Ktor
A mature family of frameworks with an established ecosystem that is used by millions of developers worldwide. • Create a RESTful web service with Spring Boot. • Build web applications with Spring Boot and Kotlin. • Use Spring Boot with Kotlin and RSocket.	A lightweight framework for those who value freedom in making architectural decisions. • Create HTTP APIs with Ktor. • Create a WebSocket chat with Ktor. • Create an interactive website with Ktor. • Publish server-side Kotlin applications: Ktor on Heroku.

2. Use Kotlin and third-party libraries in your application. Learn more about adding library and tool dependencies to your project.
   - The Kotlin standard library offers a lot of useful things such as collections or coroutines.
   - Take a look at the following third-party frameworks, libs and tools for Kotlin.
3. Learn more about Kotlin for server-side:
   - How to write your first unit test.
   - How to mix Kotlin and Java code in your application.
4. Join the Kotlin server-side community:
   - Slack: get an invite and join the #getting-started, #server, #spring, or #ktor channels.
   - StackOverflow: subscribe to the "kotlin", "spring-kotlin", or "ktor" tags.
5. Follow Kotlin on Twitter, Reddit, and Youtube, and don't miss any important ecosystem updates.

If you've encountered any difficulties or problems, report an issue to our issue tracker.

Cross-platform mobile app

Here you'll learn how to develop and improve your cross-platform mobile application using Kotlin Multiplatform.

1. Set up your environment for cross-platform mobile development.
2. Create your first application for iOS and Android:

- To start from scratch, create a basic cross-platform mobile application with the project wizard.
- If you have an existing Android application and want to make it cross-platform, complete the Make your Android application work on iOS tutorial.
- If you prefer real-life examples, clone and play with an existing project, for example the networking and data storage project from the Create a multiplatform app using Ktor and SQLdelight tutorial or any sample project.

3. Use a wide set of multiplatform libraries to implement the required business logic only once in the shared module. Learn more about adding dependencies.

Library	Details
Ktor	Docs
Serialization	Docs and sample
Coroutines	Docs and sample
DateTime	Docs
SQLDelight	Third-party library. Docs

> You can also find a multiplatform library in the community-driven list.

4. Learn more about Kotlin Multiplatform for mobile:
   - Learn more about Kotlin Multiplatform.
   - Look through samples on GitHub.
   - Create and publish a multiplatform library.
   - Learn how Kotlin Multiplatform is used at Netflix, VMware, Yandex, and many other companies.
5. Join the Kotlin Multiplatform community:
   - Slack: get an invite and join the #getting-started and #multiplatform channels.
   - StackOverflow: Subscribe to the "kotlin-multiplatform" tag.
6. Follow Kotlin on Twitter, Reddit, and Youtube, and don't miss any important ecosystem updates.

If you've encountered any difficulties or problems, report an issue to our issue tracker.

Android app

- If you want to start using Kotlin for Android development, read Google's recommendation for getting started with Kotlin on Android.
- If you're new to Android and want to learn to create applications with Kotlin, check out this Udacity course.

Follow Kotlin on Twitter, Reddit, and Youtube, and don't miss any important ecosystem updates.

Multiplatform library

Support for multiplatform programming is one of Kotlin's key benefits. It reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming.

Here you'll learn how to develop and publish a multiplatform library:

1. Create a multiplatform library:
   - Complete the Create and publish a multiplatform library tutorial. It shows how to create a multiplatform library for JVM, JS, and Native platforms, test it and publish to a local Maven repository.
2. Use libraries in your application:
   - Ktor
   - Serialization
   - Coroutines

* DateTime

> Learn more about adding dependencies on libraries. You can also find a multiplatform library in the community-driven list.

3. Learn more about Kotlin Multiplatform programming:
   * Introduction to Kotlin Multiplatform.
   * Kotlin Multiplatform supported platforms.
   * Kotlin Multiplatform programming benefits.
4. Join the Kotlin Multiplatform community:
   * Slack: get an invite and join the #getting-started and #multiplatform channels.
   * StackOverflow: Subscribe to the "kotlin-multiplatform" tag.
5. Follow Kotlin on Twitter, Reddit, and Youtube, and don't miss any important ecosystem updates.

If you've encountered any difficulties or problems, report an issue to our issue tracker.

Is anything missing?

If anything is missing or seems confusing on this page, please share your feedback.

Welcome to our tour of Kotlin!

> This tour covers the fundamentals of the Kotlin programming language and can be completed entirely within your browser. There is no installation required.

Each chapter in this tour contains:

* Theory to introduce key concepts of the language with examples.
* Practice with exercises to test your understanding of what you have learned.
* Solutions for your reference.

In this tour you will learn about:

* Variables
* Basic types
* Collections
* Control flow
* Functions
* Classes
* Null safety

To have the best experience, we recommend that you read through these chapters in order. But if you want, you can choose which chapters you want to read.

Ready to go?

Start the Kotlin tour

Hello world

Here is a simple program that prints "Hello, world!":

```
fun main() {
    println("Hello, world!")
    // Hello, world!
}
```

In Kotlin:

- fun is used to declare a function
- the main() function is where your program starts from
- the body of a function is written within curly braces {}
- println() and print() functions print their arguments to standard output

> Functions are discussed in more detail in a couple of chapters. Until then, all examples use the main() function.

Variables

All programs need to be able to store data, and variables help you to do just that. In Kotlin, you can declare:

- read-only variables with val
- mutable variables with var

To assign a value, use the assignment operator =.

For example:

```
fun main() {
//sampleStart
    val popcorn = 5    // There are 5 boxes of popcorn
    val hotdog = 7     // There are 7 hotdogs
    var customers = 10 // There are 10 customers in the queue

    // Some customers leave the queue
    customers = 8
//sampleEnd
}
```

> Variables can be declared outside the main() function at the beginning of your program. Variables declared in this way are said to be declared at top level.

As customers is a mutable variable, its value can be reassigned after declaration.

> We recommend that you declare all variables as read-only (val) by default. Declare mutable variables (var) only if necessary.

String templates

It's useful to know how to print the contents of variables to standard output. You can do this with string templates. You can use template expressions to access data stored in variables and other objects, and convert them into strings. A string value is a sequence of characters in double quotes ". Template expressions always start with a dollar sign $.

To evaluate a piece of code in a template expression, place the code within curly braces {} after the dollar sign $.

For example:

```
fun main() {
//sampleStart
    val customers = 10
```

```
    println("There are $customers customers")
    // There are 10 customers

    println("There are ${customers + 1} customers")
    // There are 11 customers
//sampleEnd
}
```

For more information, see String templates.

You will notice that there aren't any types declared for variables. Kotlin has inferred the type itself: Int. This tour explains the different Kotlin basic types and how to declare them in the next chapter.

Practice

Exercise

Complete the code to make the program print "Mary is 20 years old" to standard output:

```
fun main() {
    val name = "Mary"
    val age = 20
    // Write your code here
}
```

fun main() { val name = "Mary" val age = 20 println("$name is $age years old") }

Next step

Basic types

Basic types

Every variable and data structure in Kotlin has a data type. Data types are important because they tell the compiler what you are allowed to do with that variable or data structure. In other words, what functions and properties it has.

In the last chapter, Kotlin was able to tell in the previous example that customers has type: Int. Kotlin's ability to infer the data type is called type inference. customers is assigned an integer value. From this, Kotlin infers that customers has numerical data type: Int. As a result, the compiler knows that you can perform arithmetic operations with customers:

```
fun main() {
//sampleStart
    var customers = 10

    // Some customers leave the queue
    customers = 8

    customers = customers + 3 // Example of addition: 11
    customers += 7            // Example of addition: 18
    customers -= 3            // Example of subtraction: 15
    customers *= 2            // Example of multiplication: 30
    customers /= 3            // Example of division: 10

    println(customers) // 10
//sampleEnd
}
```

> +=, -=, *=, /=, and %= are augmented assignment operators. For more information, see Augmented assignments.

In total, Kotlin has the following basic types:

Category	Basic types
Integers	Byte, Short, Int, Long
Unsigned integers	UByte, UShort, UInt, ULong
Floating-point numbers	Float, Double
Booleans	Boolean
Characters	Char
Strings	String

For more information on basic types and their properties, see Basic types.

With this knowledge, you can declare variables and initialize them later. Kotlin can manage this as long as variables are initialized before the first read.

To declare a variable without initializing it, specify its type with :.

For example:

```
fun main() {
//sampleStart
    // Variable declared without initialization
    val d: Int
    // Variable initialized
    d = 3

    // Variable explicitly typed and initialized
    val e: String = "hello"

    // Variables can be read because they have been initialized
    println(d) // 3
    println(e) // hello
//sampleEnd
}
```

Now that you know how to declare basic types, it's time to learn about collections.

Practice

Exercise

Explicitly declare the correct type for each variable:

```
fun main() {
    val a = 1000
    val b = "log message"
    val c = 3.14
    val d = 100_000_000_000_000
    val e = false
    val f = '\n'
}
```

fun main() { val a: Int = 1000 val b: String = "log message" val c: Double = 3.14 val d: Long = 100_000_000_000_000 val e: Boolean = false val f: Char = '\n' }

Next step

[Collections](#)

Collections

When programming, it is useful to be able to group data into structures for later processing. Kotlin provides collections for exactly this purpose.

Kotlin has the following collections for grouping items:

Collection type	Description
Lists	Ordered collections of items
Sets	Unique unordered collections of items
Maps	Sets of key-value pairs where keys are unique and map to only one value

Each collection type can be mutable or read only.

List

To create a read-only list (List), use the listOf() function.

To create a mutable list (MutableList), use the mutableListOf() function.

When creating lists, Kotlin can infer the type of items stored. To declare the type explicitly, add the type within angled brackets <> after the list declaration:

```
fun main() {
//sampleStart
    // Read only list
    val readOnlyShapes = listOf("triangle", "square", "circle")
    // Mutable list with explicit type declaration
    val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")
//sampleEnd
}
```

> To prevent unwanted modifications, you can obtain read-only views of mutable lists by assigning them to a List:
>
> ```
> val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")
> val shapesLocked: List<String> = shapes
> ```
>
> This is also called casting.

Lists are ordered so to access an item in a list, use the indexed access operator []:

```
fun main() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("The first item in the list is: ${readOnlyShapes[0]}")
    // The first item in the list is: triangle
//sampleEnd
}
```

To get the first or last item in a list, use .first() and .last() functions respectively:

```kotlin
fun main() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("The first item in the list is: ${readOnlyShapes.first()}")
    // The first item in the list is: triangle
//sampleEnd
}
```

> .first() and .last() functions are examples of extension functions. To call an extension function on an object, write the function name after the object appended with a period .
>
> For more information about extension functions, see Extension functions. For the purposes of this tour, you only need to know how to call them.

To get the number of items in a list, use the .count() function:

```kotlin
fun main() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("This list has ${readOnlyShapes.count()} items")
    // This list has 3 items
//sampleEnd
}
```

To check that an item is in a list, use the in operator:

```kotlin
fun main() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("circle" in readOnlyShapes)
    // true
//sampleEnd
}
```

To add or remove items from a mutable list, use .add() and .remove() functions respectively:

```kotlin
fun main() {
//sampleStart
    val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")
    // Add "pentagon" to the list
    shapes.add("pentagon")
    println(shapes)
    // [triangle, square, circle, pentagon]

    // Remove the first "pentagon" from the list
    shapes.remove("pentagon")
    println(shapes)
    // [triangle, square, circle]
//sampleEnd
}
```

Set

To create a read-only set (Set), use the setOf() function.

To create a mutable set (MutableSet), use the mutableSetOf() function.

When creating sets, Kotlin can infer the type of items stored. To declare the type explicitly, add the type within angled brackets <> after the set declaration:

```kotlin
fun main() {
//sampleStart
    // Read-only set
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    // Mutable set with explicit type declaration
    val fruit: MutableSet<String> = mutableSetOf("apple", "banana", "cherry", "cherry")

    println(readOnlyFruit)
    // [apple, banana, cherry]
```

```
//sampleEnd
}
```

You can see in the previous example that because sets only contain unique elements, the duplicate "cherry" item is dropped.

To prevent unwanted modifications, obtain read-only views of mutable sets by casting them to Set:

```
val fruit: MutableSet<String> = mutableSetOf("apple", "banana", "cherry", "cherry")
val fruitLocked: Set<String> = fruit
```

As sets are unordered, you can't access an item at a particular index.

To get the number of items in a set, use the .count() function:

```
fun main() {
//sampleStart
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    println("This set has ${readOnlyFruit.count()} items")
    // This set has 3 items
//sampleEnd
}
```

To check that an item is in a set, use the in operator:

```
fun main() {
//sampleStart
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    println("banana" in readOnlyFruit)
    // true
//sampleEnd
}
```

To add or remove items from a mutable set, use .add() and .remove() functions respectively:

```
fun main() {
//sampleStart
    val fruit: MutableSet<String> = mutableSetOf("apple", "banana", "cherry", "cherry")
    fruit.add("dragonfruit")    // Add "dragonfruit" to the set
    println(fruit)              // [apple, banana, cherry, dragonfruit]

    fruit.remove("dragonfruit") // Remove "dragonfruit" from the set
    println(fruit)              // [apple, banana, cherry]
//sampleEnd
}
```

Map

To create a read-only map (Map), use the mapOf() function.

To create a mutable map (MutableMap), use the mutableMapOf() function.

When creating maps, Kotlin can infer the type of items stored. To declare the type explicitly, add the types of the keys and values within angled brackets <> after the map declaration.

The easiest way to create maps is to use to between each key and its related value:

```
fun main() {
//sampleStart
    // Read-only map
    val readOnlyAccountBalances = mapOf(1 to 100, 2 to 100, 3 to 100)
    // Mutable map with explicit type declaration
    val accountBalances: MutableMap<Int, Int> = mutableMapOf(1 to 100, 2 to 100, 3 to 100)
//sampleEnd
```

```
}
```

To prevent unwanted modifications, obtain read-only views of mutable maps by casting them to Map:

```
val accountBalances: MutableMap<Int, Int> = mutableMapOf(1 to 100, 2 to 100, 3 to 100)
val accountBalancesLocked: Map<Int, Int> = accountBalances
```

To access a value in a map, use the indexed access operator [] with its key:

```
fun main() {
//sampleStart
    val readOnlyAccountBalances = mapOf(1 to 100, 2 to 100, 3 to 100)
    println("The first value in the map is: ${readOnlyAccountBalances[1]}")
    // The first value in the map is: 100
//sampleEnd
}
```

To get the number of items in a map, use the .count() function:

```
fun main() {
//sampleStart
    val readOnlyAccountBalances = mapOf(1 to 100, 2 to 100, 3 to 100)
    println("This map has ${readOnlyAccountBalances.count()} key-value pairs")
    // This map has 3 key-value pairs
//sampleEnd
}
```

To add or remove items from a mutable map, use .put() and .remove() functions respectively:

```
fun main() {
//sampleStart
    val accountBalances: MutableMap<Int, Int> = mutableMapOf(1 to 100, 2 to 100, 3 to 100)
    accountBalances.put(4, 100)  // Add key 4 with value 100 to the list
    println(accountBalances)     // {1=100, 2=100, 3=100, 4=100}

    accountBalances.remove(4)    // Remove the key 4 from the list
    println(accountBalances)     // {1=100, 2=100, 3=100}
//sampleEnd
}
```

To check if a specific key is already included in a map, use the .containsKey() function:

```
fun main() {
//sampleStart
    val readOnlyAccountBalances = mapOf(1 to 100, 2 to 100, 3 to 100)
    println(readOnlyAccountBalances.containsKey(2))
    // true
//sampleEnd
}
```

To obtain a collection of the keys or values of a map, use the keys and values properties respectively:

```
fun main() {
//sampleStart
    val readOnlyAccountBalances = mapOf(1 to 100, 2 to 100, 3 to 100)
    println(readOnlyAccountBalances.keys)
    // [1, 2, 3]
    println(readOnlyAccountBalances.values)
    // [100, 100, 100]
//sampleEnd
}
```

keys and values are examples of properties of an object. To access the property of an object, write the property name after the object appended with a period .

Properties are discussed in more detail in the Classes chapter. At this point in the tour, you only need to know how to access them.

To check that a key or value is in a map, use the in operator:

```
fun main() {
//sampleStart
    val readOnlyAccountBalances = mapOf(1 to 100, 2 to 100, 3 to 100)
    println(2 in readOnlyAccountBalances.keys)
    // true
    println(200 in readOnlyAccountBalances.values)
    // false
//sampleEnd
}
```

For more information on what you can do with collections, see Collections.

Now that you know about basic types and how to manage collections, it's time to explore the control flow that you can use in your programs.

Practice

Exercise 1

You have a list of "green" numbers and a list of "red" numbers. Complete the code to print how many numbers there are in total.

```
fun main() {
    val greenNumbers = listOf(1, 4, 23)
    val redNumbers = listOf(17, 2)
    // Write your code here
}
```

fun main() { val greenNumbers = listOf(1, 4, 23) val redNumbers = listOf(17, 2) val totalCount = greenNumbers.count() + redNumbers.count() println(totalCount) }

Exercise 2

You have a set of protocols supported by your server. A user requests to use a particular protocol. Complete the program to check whether the requested protocol is supported or not (isSupported must be a Boolean value).

```
fun main() {
    val SUPPORTED = setOf("HTTP", "HTTPS", "FTP")
    val requested = "smtp"
    val isSupported = // Write your code here
    println("Support for $requested: $isSupported")
}
```

Hint
Make sure that you check the requested protocol in upper case. You can use the .uppercase() function to help you with this.

fun main() { val SUPPORTED = setOf("HTTP", "HTTPS", "FTP") val requested = "smtp" val isSupported = requested.uppercase() in SUPPORTED println("Support for $requested: $isSupported") }

Exercise 3

Define a map that relates integer numbers from 1 to 3 to their corresponding spelling. Use this map to spell the given number.

```
fun main() {
    val number2word = // Write your code here
    val n = 2
    println("$n is spelt as '${<Write your code here >}'")
```

```
}
```

fun main() { val number2word = mapOf(1 to "one", 2 to "two", 3 to "three") val n = 2 println("$n is spelt as '${number2word[n]}'") }

Next step

Control flow

Control flow

Like other programming languages, Kotlin is capable of making decisions based on whether a piece of code is evaluated to be true. Such pieces of code are called conditional expressions. Kotlin is also able to create and iterate through loops.

Conditional expressions

Kotlin provides if and when for checking conditional expressions.

> If you have to choose between if and when, we recommend using when as it leads to more robust and safer programs.

If

To use if, add the conditional expression within parentheses () and the action to take if the result is true within curly braces {}:

```
fun main() {
//sampleStart
    val d: Int
    val check = true

    if (check) {
        d = 1
    } else {
        d = 2
    }

    println(d)
    // 1
//sampleEnd
}
```

There is no ternary operator condition ? then : else in Kotlin. Instead, if can be used as an expression. When using if as an expression, there are no curly braces {}:

```
fun main() {
//sampleStart
    val a = 1
    val b = 2

    println(if (a > b) a else b) // Returns a value: 2
//sampleEnd
}
```

When

Use when when you have a conditional expression with multiple branches. when can be used either as a statement or as an expression.

Here is an example of using when as a statement:

- Place the conditional expression within parentheses () and the actions to take within curly braces {}.
- Use -> in each branch to separate each condition from each action.

```
fun main() {
```

```
//sampleStart
    val obj = "Hello"

    when (obj) {
        // Checks whether obj equals to "1"
        "1" -> println("One")
        // Checks whether obj equals to "Hello"
        "Hello" -> println("Greeting")
        // Default statement
        else -> println("Unknown")
    }
    // Greeting
//sampleEnd
}
```

Note that all branch conditions are checked sequentially until one of them is satisfied. So only the first suitable branch is executed.

Here is an example of using when as an expression. The when syntax is assigned immediately to a variable:

```
fun main() {
//sampleStart
    val obj = "Hello"

    val result = when (obj) {
        // If obj equals "1", sets result to "one"
        "1" -> "One"
        // If obj equals "Hello", sets result to "Greeting"
        "Hello" -> "Greeting"
        // Sets result to "Unknown" if no previous condition is satisfied
        else -> "Unknown"
    }
    println(result)
    // Greeting
//sampleEnd
}
```

If when is used as an expression, the else branch is mandatory, unless the compiler can detect that all possible cases are covered by the branch conditions.

The previous example showed that when is useful for matching a variable. when is also useful when you need to check a chain of Boolean expressions:

```
fun main() {
//sampleStart
    val temp = 18

    val description = when {
        // If temp < 0 is true, sets description to "very cold"
        temp < 0 -> "very cold"
        // If temp < 10 is true, sets description to "a bit cold"
        temp < 10 -> "a bit cold"
        // If temp < 20 is true, sets description to "warm"
        temp < 20 -> "warm"
        // Sets description to "hot" if no previous condition is satisfied
        else -> "hot"
    }
    println(description)
    // warm
//sampleEnd
}
```

Ranges

Before talking about loops, it's useful to know how to construct ranges for loops to iterate over.

The most common way to create a range in Kotlin is to use the .. operator. For example, 1..4 is equivalent to 1, 2, 3, 4.

To declare a range that doesn't include the end value, use until. For example, 1 until 4 is equivalent to 1, 2, 3.

To declare a range in reverse order, use downTo. For example, 4 downTo 1 is equivalent to 4, 3, 2, 1.

To declare a range that increments in a step that isn't 1, use step and your desired increment value. For example, 1..5 step 2 is equivalent to 1, 3, 5.

You can also do the same with Char ranges:

- 'a'..'d' is equivalent to 'a', 'b', 'c', 'd'
- 'z' downTo 's' step 2 is equivalent to 'z', 'x', 'v', 't'

Loops

The two most common loop structures in programming are for and while. Use for to iterate over a range of values and perform an action. Use while to continue an action until a particular condition is satisfied.

For

Using your new knowledge of ranges, you can create a for loop that iterates over numbers 1 to 5 and prints the number each time.

Place the iterator and range within parentheses () with keyword in. Add the action you want to complete within curly braces {}:

```
fun main() {
//sampleStart
    for (number in 1..5) {
        // number is the iterator and 1..5 is the range
        print(number)
    }
    // 12345
//sampleEnd
}
```

Collections can also be iterated over by loops:

```
fun main() {
//sampleStart
    val cakes = listOf("carrot", "cheese", "chocolate")

    for (cake in cakes) {
        println("Yummy, it's a $cake cake!")
    }
    // Yummy, it's a carrot cake!
    // Yummy, it's a cheese cake!
    // Yummy, it's a chocolate cake!
//sampleEnd
}
```

While

while can be used in two ways:

- To execute a code block while a conditional expression is true. (while)
- To execute the code block first and then check the conditional expression. (do-while)

In the first use case (while):

- Declare the conditional expression for your while loop to continue within parentheses ().
- Add the action you want to complete within curly braces {}.

> The following examples use the increment operator ++ to increment the value of the cakesEaten variable.

```
fun main() {
//sampleStart
    var cakesEaten = 0
    while (cakesEaten < 3) {
        println("Eat a cake")
        cakesEaten++
    }
    // Eat a cake
    // Eat a cake
```

```
    // Eat a cake
//sampleEnd
}
```

In the second use case (do-while):

- Declare the conditional expression for your while loop to continue within parentheses ().
- Define the action you want to complete within curly braces {} with the keyword do.

```
fun main() {
//sampleStart
    var cakesEaten = 0
    var cakesBaked = 0
    while (cakesEaten < 3) {
        println("Eat a cake")
        cakesEaten++
    }
    do {
        println("Bake a cake")
        cakesBaked++
    } while (cakesBaked < cakesEaten)
    // Eat a cake
    // Eat a cake
    // Eat a cake
    // Bake a cake
    // Bake a cake
    // Bake a cake
//sampleEnd
}
```

For more information and examples of conditional expressions and loops, see Conditions and loops.

Now that you know the fundamentals of Kotlin control flow, it's time to learn how to write your own functions.

Practice

Exercise 1

Using a when expression, update the following program so that when you input the names of GameBoy buttons, the actions are printed to output.

Button	Action
A	Yes
B	No
X	Menu
Y	Nothing
Other	There is no such button

```
fun main() {
    val button = "A"

    println(
        // Write your code here
    )
}
```

fun main() { val button = "A" println(when (button) { "A" -> "Yes" "B" -> "No" "X" -> "Menu" "Y" -> "Nothing" else -> "There is no such button" }) }

Exercise 2

You have a program that counts pizza slices until there's a whole pizza with 8 slices. Refactor this program in two ways:

- Use a while loop.
- Use a do-while loop.

```
fun main() {
    var pizzaSlices = 0
    // Start refactoring here
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    // End refactoring here
    println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D")
}
```

fun main() { var pizzaSlices = 0 while (pizzaSlices < 7) { pizzaSlices++ println("There's only $pizzaSlices slice/s of pizza :(") } pizzaSlices++ println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D") }

fun main() { var pizzaSlices = 0 pizzaSlices++ do { println("There's only $pizzaSlices slice/s of pizza :(") pizzaSlices++ } while (pizzaSlices < 8) println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D") }

Exercise 3

Write a program that simulates the Fizz buzz game. Your task is to print numbers from 1 to 100 incrementally, replacing any number divisible by three with the word "fizz", and any number divisible by five with the word "buzz". Any number divisible by both 3 and 5 must be replaced with the word "fizzbuzz".

Hint

Use a for loop to count numbers and a when expression to decide what to print at each step.

```
fun main() {
    // Write your code here
}
```

fun main() { for (number in 1..100) { println(when { number % 15 == 0 -> "fizzbuzz" number % 3 == 0 -> "fizz" number % 5 == 0 -> "buzz" else -> number.toString() }) } }

Exercise 4

You have a list of words. Use for and if to print only the words that start with the letter l.

Hint

Use the .startsWith() function for String type.

```
fun main() {
    val words = listOf("dinosaur", "limousine", "magazine", "language")
    // Write your code here
}
```

fun main() { val words = listOf("dinosaur", "limousine", "magazine", "language") for (w in words) { if (w.startsWith("l")) println(w) } }

Next step

Functions

Functions

You can declare your own functions in Kotlin using the fun keyword.

```
fun hello(){
    return println("Hello, world!")
}

fun main() {
    hello()
    // Hello, world!
}
```

In Kotlin:

- function parameters are written within parentheses ().
- each parameter must have a type, and multiple parameters must be separated by commas ,.
- the return type is written after the function's parentheses (), separated by a colon :.
- the body of a function is written within curly braces {}.
- the return keyword is used to exit or return something from a function.

> If a function doesn't return anything useful, the return type and return keyword can be omitted. Learn more about this in Functions without return.

In the following example:

- x and y are function parameters.
- x and y have type Int.
- the function's return type is Int.
- the function returns a sum of x and y when called.

```
fun sum(x: Int, y: Int): Int {
    return x + y
}

fun main() {
    println(sum(1, 2))
    // 3
}
```

> We recommend in our coding conventions that you name functions starting with a lowercase letter and use camel case with no underscores.

Named arguments

For concise code, when calling your function, you don't have to include parameter names. However, including parameter names does make your code easier to read. This is called using named arguments. If you do include parameter names, then you can write the parameters in any order.

> In the following example, string templates ($) are used to access the parameter values, convert them to String type, and then concatenate them into a string for printing.

```kotlin
fun printMessageWithPrefix(message: String, prefix: String = "Info") {
    println("[$prefix] $message")
}

fun main() {
    // Uses named arguments with swapped parameter order
    printMessageWithPrefix(prefix = "Log", message = "Hello")
    // [Log] Hello
}
```

Default parameter values

You can define default values for your function parameters. Any parameter with a default value can be omitted when calling your function. To declare a default value, use the assignment operator = after the type:

```kotlin
fun printMessageWithPrefix(message: String, prefix: String = "Info") {
    println("[$prefix] $message")
}

fun main() {
    // Function called with both parameters
    printMessageWithPrefix("Hello", "Log")
    // [Log] Hello

    // Function called only with message parameter
    printMessageWithPrefix("Hello")
    // [Info] Hello

    printMessageWithPrefix(prefix = "Log", message = "Hello")
    // [Log] Hello
}
```

> You can skip specific parameters with default values, rather than omitting them all. However, after the first skipped parameter, you must name all subsequent parameters.

Functions without return

If your function doesn't return a useful value then its return type is Unit. Unit is a type with only one value – Unit. You don't have to declare that Unit is returned explicitly in your function body. This means that you don't have to use the return keyword or declare a return type:

```kotlin
fun printMessage(message: String) {
    println(message)
    // `return Unit` or `return` is optional
}

fun main() {
    printMessage("Hello")
    // Hello
}
```

Single-expression functions

To make your code more concise, you can use single-expression functions. For example, the sum() function can be shortened:

```kotlin
fun sum(x: Int, y: Int): Int {
    return x + y
}

fun main() {
    println(sum(1, 2))
    // 3
}
```

You can remove the curly braces {} and declare the function body using the assignment operator =. And due to Kotlin's type inference, you can also omit the return

type. The sum() function then becomes one line:

```
fun sum(x: Int, y: Int) = x + y

fun main() {
    println(sum(1, 2))
    // 3
}
```

Omitting the return type is only possible when your function has no body ({}). Unless your function's return type is Unit.

Functions practice

Exercise 1

Write a function called circleArea that takes the radius of a circle in integer format as a parameter and outputs the area of that circle.

In this exercise, you import a package so that you can access the value of pi via PI. For more information about importing packages, see Packages and imports.

```
import kotlin.math.PI

fun circleArea() {
    // Write your code here
}
fun main() {
    println(circleArea(2))
}
```

import kotlin.math.PI fun circleArea(radius: Int): Double { return PI * radius * radius } fun main() { println(circleArea(2)) // 12.566370614359172 }

Exercise 2

Rewrite the circleArea function from the previous exercise as a single-expression function.

```
import kotlin.math.PI

// Write your code here

fun main() {
    println(circleArea(2))
}
```

import kotlin.math.PI fun circleArea(radius: Int): Double = PI * radius * radius fun main() { println(circleArea(2)) // 12.566370614359172 }

Exercise 3

You have a function that translates a time interval given in hours, minutes, and seconds into seconds. In most cases, you need to pass only one or two function parameters while the rest are equal to 0. Improve the function and the code that calls it by using default parameter values and named arguments so that the code is easier to read.

```
fun intervalInSeconds(hours: Int, minutes: Int, seconds: Int) =
    ((hours * 60) + minutes) * 60 + seconds

fun main() {
    println(intervalInSeconds(1, 20, 15))
    println(intervalInSeconds(0, 1, 25))
    println(intervalInSeconds(2, 0, 0))
```

```
    println(intervalInSeconds(0, 10, 0))
    println(intervalInSeconds(1, 0, 1))
}
```

fun intervalInSeconds(hours: Int = 0, minutes: Int = 0, seconds: Int = 0) = ((hours * 60) + minutes) * 60 + seconds fun main() { println(intervalInSeconds(1, 20, 15)) println(intervalInSeconds(minutes = 1, seconds = 25)) println(intervalInSeconds(hours = 2)) println(intervalInSeconds(minutes = 10)) println(intervalInSeconds(hours = 1, seconds = 1)) }

Lambda expressions

Kotlin allows you to write even more concise code for functions by using lambda expressions.

For example, the following uppercaseString() function:

```
fun uppercaseString(string: String): String {
    return string.uppercase()
}
fun main() {
    println(uppercaseString("hello"))
    // HELLO
}
```

Can also be written as a lambda expression:

```
fun main() {
    println({ string: String -> string.uppercase() }("hello"))
    // HELLO
}
```

Lambda expressions can be hard to understand at first glance so let's break it down. Lambda expressions are written within curly braces {}.

Within the lambda expression, you write:

- the parameters followed by an ->.
- the function body after the ->.

In the previous example:

- string is a function parameter.
- string has type String.
- the function returns the result of the .uppercase() function called on string.

> If you declare a lambda without parameters, then there is no need to use ->. For example:
>
> ```
> { println("Log message") }
> ```

Lambda expressions can be used in a number of ways. You can:

- assign a lambda to a variable that you can then invoke later
- pass a lambda expression as a parameter to another function
- return a lambda expression from a function
- invoke a lambda expression on its own

Assign to variable

To assign a lambda expression to a variable, use the assignment operator =:

```
fun main() {
```

```
    val upperCaseString = { string: String -> string.uppercase() }
    println(upperCaseString("hello"))
    // HELLO
}
```

Pass to another function

A great example of when it is useful to pass a lambda expression to a function, is using the .filter() function on collections:

```
fun main() {
    //sampleStart
    val numbers = listOf(1, -2, 3, -4, 5, -6)
    val positives = numbers.filter { x -> x > 0 }
    val negatives = numbers.filter { x -> x < 0 }
    println(positives)
    // [1, 3, 5]
    println(negatives)
    // [-2, -4, -6]
    //sampleEnd
}
```

The .filter() function accepts a lambda expression as a predicate:

- { x -> x > 0 } takes each element of the list and returns only those that are positive.
- { x -> x < 0 } takes each element of the list and returns only those that are negative.

> If a lambda expression is the only function parameter, you can drop the function parentheses (). This is an example of a trailing lambda, which is discussed in more detail at the end of this chapter.

Another good example, is using the .map() function to transform items in a collection:

```
fun main() {
    //sampleStart
    val numbers = listOf(1, -2, 3, -4, 5, -6)
    val doubled = numbers.map { x -> x * 2 }
    val tripled = numbers.map { x -> x * 3 }
    println(doubled)
    // [2, -4, 6, -8, 10, -12]
    println(tripled)
    // [3, -6, 9, -12, 15, -18]
    //sampleEnd
}
```

The .map() function accepts a lambda expression as a predicate:

- { x -> x * 2 } takes each element of the list and returns that element multiplied by 2.
- { x -> x * 3 } takes each element of the list and returns that element multiplied by 3.

Function types

Before you can return a lambda expression from a function, you first need to understand function types.

You have already learned about basic types but functions themselves also have a type. Kotlin's type inference can infer a function's type from the parameter type. But there may be times when you need to explicitly specify the function type. The compiler needs the function type so that it knows what is and isn't allowed for that function.

The syntax for a function type has:

- each parameter's type written within parentheses () and separated by commas ,.
- the return type written after ->.

For example: (String) -> String or (Int, Int) -> Int.

This is what a lambda expression looks like if a function type for upperCaseString() is defined:

```kotlin
val upperCaseString: (String) -> String = { string -> string.uppercase() }

fun main() {
    println(upperCaseString("hello"))
    // HELLO
}
```

If your lambda expression has no parameters then the parentheses () are left empty. For example: () -> Unit

> You must declare parameter and return types either in the lambda expression or as a function type. Otherwise, the compiler won't be able to know what type your lambda expression is.
>
> For example, the following won't work:
>
> val upperCaseString = { str -> str.uppercase() }

Return from a function

Lambda expressions can be returned from a function. So that the compiler understands what type the lambda expression returned is, you must declare a function type.

In the following example, the toSeconds() function has function type (Int) -> Int because it always returns a lambda expression that takes a parameter of type Int and returns an Int value.

This example uses a when expression to determine which lambda expression is returned when toSeconds() is called:

```kotlin
fun toSeconds(time: String): (Int) -> Int = when (time) {
    "hour" -> { value -> value * 60 * 60 }
    "minute" -> { value -> value * 60 }
    "second" -> { value -> value }
    else -> { value -> value }
}

fun main() {
    val timesInMinutes = listOf(2, 10, 15, 1)
    val min2sec = toSeconds("minute")
    val totalTimeInSeconds = timesInMinutes.map(min2sec).sum()
    println("Total time is $totalTimeInSeconds secs")
    // Total time is 1680 secs
}
```

Invoke separately

Lambda expressions can be invoked on their own by adding parentheses () after the curly braces {} and including any parameters within the parentheses:

```kotlin
fun main() {
    //sampleStart
    println({ string: String -> string.uppercase() }("hello"))
    // HELLO
    //sampleEnd
}
```

Trailing lambdas

As you have already seen, if a lambda expression is the only function parameter, you can drop the function parentheses (). If a lambda expression is passed as the last parameter of a function, then the expression can be written outside the function parentheses (). In both cases, this syntax is called a trailing lambda.

For example, the .fold() function accepts an initial value and an operation:

```kotlin
fun main() {
    //sampleStart
    // The initial value is zero.
    // The operation sums the initial value with every item in the list cumulatively.
    println(listOf(1, 2, 3).fold(0, { x, item -> x + item })) // 6

    // Alternatively, in the form of a trailing lambda
    println(listOf(1, 2, 3).fold(0) { x, item -> x + item })  // 6
    //sampleEnd
```

```
}
```

For more information on lambda expressions, see Lambda expressions and anonymous functions.

The next step in our tour is to learn about classes in Kotlin.

Lambda expressions practice

Exercise 1

You have a list of actions supported by a web service, a common prefix for all requests, and an ID of a particular resource. To request an action title over the resource with ID: 5, you need to create the following URL: https://example.com/book-info/5/title. Use a lambda expression to create a list of URLs from the list of actions.

```
fun main() {
    val actions = listOf("title", "year", "author")
    val prefix = "https://example.com/book-info"
    val id = 5
    val urls = // Write your code here
        println(urls)
}
```

fun main() { val actions = listOf("title", "year", "author") val prefix = "https://example.com/book-info" val id = 5 val urls = actions.map { action -> "$prefix/$id/$action" } println(urls) }

Exercise 2

Write a function that takes an Int value and an action (a function with type () -> Unit) which then repeats the action the given number of times. Then use this function to print “Hello” 5 times.

```
fun repeatN(n: Int, action: () -> Unit) {
    // Write your code here
}

fun main() {
    // Write your code here
}
```

fun repeatN(n: Int, action: () -> Unit) { for (i in 1..n) { action() } } fun main() { repeatN(5) { println("Hello") } }

Next step

Classes

Classes

Kotlin supports object-oriented programming with classes and objects. Objects are useful for storing data in your program. Classes allow you to declare a set of characteristics for an object. When you create objects from a class, you can save time and effort because you don't have to declare these characteristics every time.

To declare a class, use the class keyword:

```
class Customer
```

Properties

Characteristics of a class's object can be declared in properties. You can declare properties for a class:

* Within parentheses () after the class name.

```
class Contact(val id: Int, var email: String)
```

* Within the class body defined by curly braces {}.

```
class Contact(val id: Int, var email: String) {
    val category: String = ""
}
```

We recommend that you declare properties as read-only (val) unless they need to be changed after an instance of the class is created.

You can declare properties without val or var within parentheses but these properties are not accessible after an instance has been created.

> * The content contained within parentheses () is called the class header.
> * You can use a trailing comma when declaring class properties.

Just like with function parameters, class properties can have default values:

```
class Contact(val id: Int, var email: String = "example@gmail.com") {
    val category: String = "work"
}
```

Create instance

To create an object from a class, you declare a class instance using a constructor.

By default, Kotlin automatically creates a constructor with the parameters declared in the class header.

For example:

```
class Contact(val id: Int, var email: String)

fun main() {
    val contact = Contact(1, "mary@gmail.com")
}
```

In the example:

* Contact is a class.
* contact is an instance of the Contact class.
* id and email are properties.
* id and email are used with the default constructor to create contact.

Kotlin classes can have many constructors, including ones that you define yourself. To learn more about how to declare multiple constructors, see Constructors.

Access properties

To access a property of an instance, write the name of the property after the instance name appended with a period .:

```
class Contact(val id: Int, var email: String)

fun main() {
    val contact = Contact(1, "mary@gmail.com")

    // Prints the value of the property: email
    println(contact.email)
```

```
    // mary@gmail.com

    // Updates the value of the property: email
    contact.email = "jane@gmail.com"

    // Prints the new value of the property: email
    println(contact.email)
    // jane@gmail.com
}
```

To concatenate the value of a property as part of a string, you can use string templates ($). For example:

```
println("Their email address is: ${contact.email}")
```

Member functions

In addition to declaring properties as part of an object's characteristics, you can also define an object's behavior with member functions.

In Kotlin, member functions must be declared within the class body. To call a member function on an instance, write the function name after the instance name appended with a period .. For example:

```
class Contact(val id: Int, var email: String) {
    fun printId() {
        println(id)
    }
}

fun main() {
    val contact = Contact(1, "mary@gmail.com")
    // Calls member function printId()
    contact.printId()
    // 1
}
```

Data classes

Kotlin has data classes which are particularly useful for storing data. Data classes have the same functionality as classes, but they come automatically with additional member functions. These member functions allow you to easily print the instance to readable output, compare instances of a class, copy instances, and more. As these functions are automatically available, you don't have to spend time writing the same boilerplate code for each of your classes.

To declare a data class, use the keyword data:

```
data class User(val name: String, val id: Int)
```

The most useful predefined member functions of data classes are:

Function	Description
.toString()	Prints a readable string of the class instance and its properties.
.equals() or ==	Compares instances of a class.
.copy()	Creates a class instance by copying another, potentially with some different properties.

See the following sections for examples of how to use each function:

- Print as string

* Compare instances
* Copy instance

Print as string

To print a readable string of a class instance, you can explicitly call the .toString() function, or use print functions (println() and print()) which automatically call .toString() for you:

```kotlin
data class User(val name: String, val id: Int)

fun main() {
    val user = User("Alex", 1)

    //sampleStart
    // Automatically uses toString() function so that output is easy to read
    println(user)
    // User(name=Alex, id=1)
    //sampleEnd
}
```

This is particularly useful when debugging or creating logs.

Compare instances

To compare data class instances, use the equality operator ==:

```kotlin
data class User(val name: String, val id: Int)

fun main() {
    //sampleStart
    val user = User("Alex", 1)
    val secondUser = User("Alex", 1)
    val thirdUser = User("Max", 2)

    // Compares user to second user
    println("user == secondUser: ${user == secondUser}")
    // user == secondUser: true

    // Compares user to third user
    println("user == thirdUser: ${user == thirdUser}")
    // user == thirdUser: false
    //sampleEnd
}
```

Copy instance

To create an exact copy of a data class instance, call the .copy() function on the instance.

To create a copy of a data class instance and change some properties, call the .copy() function on the instance and add replacement values for properties as function parameters.

For example:

```kotlin
data class User(val name: String, val id: Int)

fun main() {
    //sampleStart
    val user = User("Alex", 1)
    val secondUser = User("Alex", 1)
    val thirdUser = User("Max", 2)

    // Creates an exact copy of user
    println(user.copy())
    // User(name=Alex, id=1)

    // Creates a copy of user with name: "Max"
    println(user.copy("Max"))
    // User(name=Max, id=1)

    // Creates a copy of user with id: 3
    println(user.copy(id = 3))
```

```
    // User(name=Alex, id=3)
    //sampleEnd
}
```

Creating a copy of an instance is safer than modifying the original instance because any code that relies on the original instance isn't affected by the copy and what you do with it.

For more information about data classes, see Data classes.

The last chapter of this tour is about Kotlin's null safety.

Practice

Exercise 1

Define a data class Employee with two properties: one for a name, and another for a salary. Make sure that the property for salary is mutable, otherwise you won't get a salary boost at the end of the year! The main function demonstrates how you can use this data class.

```
// Write your code here

fun main() {
    val emp = Employee("Mary", 20)
    println(emp)
    emp.salary += 10
    println(emp)
}
```

data class Employee(val name: String, var salary: Int) fun main() { val emp = Employee("Mary", 20) println(emp) emp.salary += 10 println(emp) }

Exercise 2

To test your code, you need a generator that can create random employees. Define a class with a fixed list of potential names (inside the class body), and that is configured by a minimum and maximum salary (inside the class header). Once again, the main function demonstrates how you can use this class.

Hint
Lists have an extension function called .random() that returns a random item within a list.

Hint
Random.nextInt(from = ..., until = ...) gives you a random Int number within specified limits.

```
import kotlin.random.Random

data class Employee(val name: String, var salary: Int)

// Write your code here

fun main() {
    val empGen = RandomEmployeeGenerator(10, 30)
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    empGen.minSalary = 50
    empGen.maxSalary = 100
    println(empGen.generateEmployee())
}
```

import kotlin.random.Random data class Employee(val name: String, var salary: Int) class RandomEmployeeGenerator(var minSalary: Int, var maxSalary: Int) { val names = listOf("John", "Mary", "Ann", "Paul", "Jack", "Elizabeth") fun generateEmployee() = Employee(names.random(), Random.nextInt(from = minSalary, until = maxSalary)) } fun main() { val empGen = RandomEmployeeGenerator(10, 30) println(empGen.generateEmployee()) println(empGen.generateEmployee()) println(empGen.generateEmployee()) empGen.minSalary = 50 empGen.maxSalary = 100 println(empGen.generateEmployee()) }

Next step

Null safety

Null safety

In Kotlin, it's possible to have a null value. To help prevent issues with null values in your programs, Kotlin has null safety in place. Null safety detects potential problems with null values at compile time, rather than at run time.

Null safety is a combination of features that allow you to:

- explicitly declare when null values are allowed in your program.
- check for null values.
- use safe calls to properties or functions that may contain null values.
- declare actions to take if null values are detected.

Nullable types

Kotlin supports nullable types which allows the possibility for the declared type to have null values. By default, a type is not allowed to accept null values. Nullable types are declared by explicitly adding ? after the type declaration.

For example:

```
fun main() {
    // neverNull has String type
    var neverNull: String = "This can't be null"

    // Throws a compiler error
    neverNull = null

    // nullable has nullable String type
    var nullable: String? = "You can keep a null here"

    // This is OK
    nullable = null

    // By default, null values aren't accepted
    var inferredNonNull = "The compiler assumes non-null"

    // Throws a compiler error
    inferredNonNull = null

    // notNull doesn't accept null values
    fun strLength(notNull: String): Int {
        return notNull.length
    }

    println(strLength(neverNull)) // 18
    println(strLength(nullable))  // Throws a compiler error
}
```

length is a property of the String class that contains the number of characters within a string.

Check for null values

You can check for the presence of null values within conditional expressions. In the following example, the describeString() function has an if statement that checks whether maybeString is not null and if its length is greater than zero:

```
fun describeString(maybeString: String?): String {
    if (maybeString != null && maybeString.length > 0) {
        return "String of length ${maybeString.length}"
    } else {
        return "Empty or null string"
    }
}

fun main() {
    var nullString: String? = null
    println(describeString(nullString))
```

```
    // Empty or null string
}
```

Use safe calls

To safely access properties of an object that might contain a null value, use the safe call operator ?.. The safe call operator returns null if the object's property is null. This is useful if you want to avoid the presence of null values triggering errors in your code.

In the following example, the lengthString() function uses a safe call to return either the length of the string or null:

```
fun lengthString(maybeString: String?): Int? = maybeString?.length

fun main() {
    var nullString: String? = null
    println(lengthString(nullString))
    // null
}
```

Safe calls can be chained so that if any property of an object contains a null value, then null is returned without an error being thrown. For example:

```
person.company?.address?.country
```

The safe call operator can also be used to safely call an extension or member function. In this case, a null check is performed before the function is called. If the check detects a null value, then the call is skipped and null is returned.

In the following example, nullString is null so the invocation of .uppercase() is skipped and null is returned:

```
fun main() {
    var nullString: String? = null
    println(nullString?.uppercase())
    // null
}
```

Use Elvis operator

You can provide a default value to return if a null value is detected by using the Elvis operator ?:.

Write on the left-hand side of the Elvis operator what should be checked for a null value. Write on the right-hand side of the Elvis operator what should be returned if a null value is detected.

In the following example, nullString is null so the safe call to access the length property returns a null value. As a result, the Elvis operator returns 0:

```
fun main() {
    var nullString: String? = null
    println(nullString?.length ?: 0)
    // 0
}
```

For more information about null safety in Kotlin, see Null safety.

Practice

Exercise

You have the employeeById function that gives you access to a database of employees of a company. Unfortunately, this function returns a value of the Employee? type, so the result can be null. Your goal is to write a function that returns the salary of an employee when their id is provided, or 0 if the employee is missing from the database.

```kotlin
data class Employee (val name: String, var salary: Int)

fun employeeById(id: Int) = when(id) {
    1 -> Employee("Mary", 20)
    2 -> null
    3 -> Employee("John", 21)
    4 -> Employee("Ann", 23)
    else -> null
}

fun salaryById(id: Int) = // Write your code here

fun main() {
    println((1..5).sumOf { id -> salaryById(id) })
}
```

data class Employee (val name: String, var salary: Int) fun employeeById(id: Int) = when(id) { 1 -> Employee("Mary", 20) 2 -> null 3 -> Employee("John", 21) 4 -> Employee("Ann", 23) else -> null } fun salaryById(id: Int) = employeeById(id)?.salary ?: 0 fun main() { println((1..5).sumOf { id -> salaryById(id) }) }

What's next?

Congratulations! Now that you have completed the Kotlin tour, check out our tutorials for popular Kotlin applications:

- Create a backend application
- Create a cross-platform application for Android and iOS

Kotlin Multiplatform

> Kotlin Multiplatform is in Beta. It is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make.

The Kotlin Multiplatform technology is designed to simplify the development of cross-platform projects. It reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming.

Kotlin Multiplatform

Kotlin Multiplatform use cases

Android and iOS applications

Sharing code between mobile platforms is a major Kotlin Multiplatform use cases. With Kotlin Multiplatform for mobile, you can build cross-platform mobile applications that share code between Android and iOS to implement networking, data storage and data validation, analytics, computations, and other application logic.

Check out the Get started with Kotlin Multiplatform for mobile and Create a multiplatform app using Ktor and SQLDelight tutorials, where you will create applications for Android and iOS that include a module with shared code for both platforms.

Thanks to Compose Multiplatform, a Kotlin-based declarative UI framework developed by JetBrains, you can also share UIs across Android and iOS to create fully cross-platform apps:

Sharing different levels and UI

Try this Compose Multiplatform mobile application template to create your own mobile application with UIs shared between both platforms.

Multiplatform libraries

Kotlin Multiplatform is also helpful for library authors. You can create a multiplatform library with common code and its platform-specific implementations for JVM, web, and native platforms. Once published, a multiplatform library can be used as a dependency in other cross-platform projects.

See the Create and publish a multiplatform library tutorial, where you will create a multiplatform library, test it, and publish it to Maven.

Desktop applications

Compose Multiplatform helps share UIs across desktop platforms like Windows, macOS, and Linux. Many applications, including the JetBrains Toolbox app, have already adopted this approach.

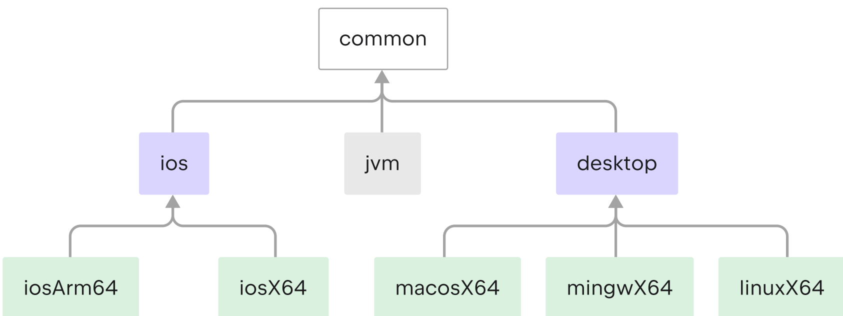
Try this Compose Multiplatform desktop application template to create your own project with UIs shared among desktop platforms.

Code sharing between platforms

Kotlin Multiplatform allows you to maintain a single codebase of the application logic for different platforms. You also get advantages of native programming, including great performance and full access to platform SDKs.

Kotlin provides the following code sharing mechanisms:

- Share common code among all platforms used in your project.
- Share code among some platforms included in your project to reuse much of the code in similar platforms:



Code shared across different platforms

- If you need to access platform-specific APIs from the shared code, use the Kotlin mechanism of expected and actual declarations.

Get started

- Begin with the Get started with Kotlin Multiplatform for mobile if you want to create iOS and Android applications with shared code
- Explore sharing code principles and examples if you want to create applications or libraries targeting other platforms

> New to Kotlin? Take a look at Getting started with Kotlin.

Sample projects

Look through cross-platform application samples to understand how Kotlin Multiplatform works:

- Kotlin Multiplatform for mobile samples
- Compose Multiplatform samples

Kotlin for server side

Kotlin is a great fit for developing server-side applications. It allows you to write concise and expressive code while maintaining full compatibility with existing Java-based technology stacks, all with a smooth learning curve:

- Expressiveness: Kotlin's innovative language features, such as its support for type-safe builders and delegated properties, help build powerful and easy-to-use abstractions.
- Scalability: Kotlin's support for coroutines helps build server-side applications that scale to massive numbers of clients with modest hardware requirements.
- Interoperability: Kotlin is fully compatible with all Java-based frameworks, so you can use your familiar technology stack while reaping the benefits of a more modern language.
- Migration: Kotlin supports gradual migration of large codebases from Java to Kotlin. You can start writing new code in Kotlin while keeping older parts of your system in Java.
- Tooling: In addition to great IDE support in general, Kotlin offers framework-specific tooling (for example, for Spring) in the plugin for IntelliJ IDEA Ultimate.
- Learning Curve: For a Java developer, getting started with Kotlin is very easy. The automated Java-to-Kotlin converter included in the Kotlin plugin helps with the first steps. Kotlin Koans can guide you through the key features of the language with a series of interactive exercises.

Frameworks for server-side development with Kotlin

Here are some examples of the server-side frameworks for Kotlin:

- Spring makes use of Kotlin's language features to offer more concise APIs, starting with version 5.0. The online project generator allows you to quickly generate a new project in Kotlin.
- Ktor is a framework built by JetBrains for creating Web applications in Kotlin, making use of coroutines for high scalability and offering an easy-to-use and idiomatic API.
- Quarkus provides first class support for using Kotlin. The framework is open source and maintained by Red Hat. Quarkus was built from the ground up for Kubernetes and provides a cohesive full-stack framework by leveraging a growing list of hundreds of best-of-breed libraries.
- Vert.x, a framework for building reactive Web applications on the JVM, offers dedicated support for Kotlin, including full documentation.
- kotlinx.html is a DSL that can be used to build HTML in Web applications. It serves as an alternative to traditional templating systems such as JSP and FreeMarker.
- Micronaut is a modern JVM-based full-stack framework for building modular, easily testable microservices and serverless applications. It comes with a lot of useful built-in features.
- http4k is the functional toolkit with a tiny footprint for Kotlin HTTP applications, written in pure Kotlin. The library is based on the "Your Server as a Function" paper from Twitter and represents modeling both HTTP servers and clients as simple Kotlin functions that can be composed together.
- Javalin is a very lightweight web framework for Kotlin and Java which supports WebSockets, HTTP2, and async requests.
- The available options for persistence include direct JDBC access, JPA, and using NoSQL databases through their Java drivers. For JPA, the kotlin-jpa compiler plugin adapts Kotlin-compiled classes to the requirements of the framework.

> You can find more frameworks at https://kotlin.link/.

Deploying Kotlin server-side applications

Kotlin applications can be deployed into any host that supports Java Web applications, including Amazon Web Services, Google Cloud Platform, and more.

To deploy Kotlin applications on Heroku, you can follow the official Heroku tutorial.

AWS Labs provides a sample project showing the use of Kotlin for writing AWS Lambda functions.

Google Cloud Platform offers a series of tutorials for deploying Kotlin applications to GCP, both for Ktor and App Engine and Spring and App engine. In addition, there is an interactive code lab for deploying a Kotlin Spring application.

Products that use Kotlin on the server side

Corda is an open-source distributed ledger platform that is supported by major banks and built entirely in Kotlin.

JetBrains Account, the system responsible for the entire license sales and validation process at JetBrains, is written in 100% Kotlin and has been running in production since 2015 with no major issues.

Next steps

- For a more in-depth introduction to the language, check out the Kotlin documentation on this site and Kotlin Koans.
- Watch a webinar "Micronaut for microservices with Kotlin" and explore a detailed guide showing how you can use Kotlin extension functions in the Micronaut framework.
- http4k provides the CLI to generate fully formed projects, and a starter repo to generate an entire CD pipeline using GitHub, Travis, and Heroku with a single bash command.
- Want to migrate from Java to Kotlin? Learn how to perform typical tasks with strings in Java and Kotlin.

Kotlin for Android

Android mobile development has been Kotlin-first since Google I/O in 2019.

Over 50% of professional Android developers use Kotlin as their primary language, while only 30% use Java as their main language. 70% of developers whose primary language is Kotlin say that Kotlin makes them more productive.

Using Kotlin for Android development, you can benefit from:

- Less code combined with greater readability. Spend less time writing your code and working to understand the code of others.
- Fewer common errors. Apps built with Kotlin are 20% less likely to crash based on Google's internal data.
- Kotlin support in Jetpack libraries. Jetpack Compose is Android's recommended modern toolkit for building native UI in Kotlin. KTX extensions add Kotlin language features, like coroutines, extension functions, lambdas, and named parameters to existing Android libraries.
- Support for multiplatform development. Kotlin Multiplatform allows development for not only Android but also iOS, backend, and web applications. Some Jetpack libraries are already multiplatform. Compose Multiplatform, JetBrains' declarative UI framework based on Kotlin and Jetpack Compose, makes it possible to share UIs across platforms – iOS, Android, desktop, and web.
- Mature language and environment. Since its creation in 2011, Kotlin has developed continuously, not only as a language but as a whole ecosystem with robust tooling. Now it's seamlessly integrated into Android Studio and is actively used by many companies for developing Android applications.
- Interoperability with Java. You can use Kotlin along with the Java programming language in your applications without needing to migrate all your code to Kotlin.
- Easy learning. Kotlin is very easy to learn, especially for Java developers.
- Big community. Kotlin has great support and many contributions from the community, which is growing all over the world. Over 95% of the top thousand Android apps use Kotlin.

Many startups and Fortune 500 companies have already developed Android applications using Kotlin, see the list on the Google website for Android developers.

To start using Kotlin for:

- Android development, read Google's documentation for developing Android apps with Kotlin.
- Developing cross-platform mobile applications, see Get started with Kotlin Multiplatform for Android and iOS.

Kotlin Wasm

> Kotlin Wasm is Experimental. It may be changed at any time. Use it only for evaluation purposes.
>
> We would appreciate your feedback on it in YouTrack.

WebAssembly (Wasm) is a binary instruction format for a stack-based virtual machine. This format is platform-independent because it runs on its own virtual machine. Wasm is designed to be fast and secure, and it can compile code from various programming languages, including Kotlin.

Kotlin/Wasm is a new compilation target for Kotlin. You can use it in your Kotlin Multiplatform projects. With Kotlin/Wasm, you can create applications that run on different environments and devices supporting WebAssembly and meeting Kotlin's requirements.

> Learn more about Kotlin/Wasm in this YouTube video.

Browser support

Almost all major browsers already support WebAssembly 1.0. To run applications built with Kotlin/Wasm in a browser, you need to enable an experimental garbage collection feature.

Learn more in Get started with Kotlin/Wasm.

Interoperability

Kotlin/Wasm allows you to both use JavaScript code and Browser API from Kotlin, and Kotlin code from JavaScript.

Learn more about Kotlin Wasm interoperability with JavaScript.

Compose Multiplatform for Web

> Web support is Experimental and may be changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in the public Slack channel #compose-web. If you face any issues, please report them on GitHub.

Compose Multiplatform for Web is based on new Kotlin/Wasm target. You can create a Kotlin Multiplatform project and experiment with sharing your mobile or desktop UIs with the web. With Compose Multiplatform for Web, you can run your code in the browser with all the benefits of WebAssembly.

How to get started

- Get started with Kotlin/Wasm in IntelliJ IDEA
- Check out the GitHub repository with Kotlin/Wasm examples

Libraries support

You can use the Kotlin standard library (stdlib) and test library (kotlin.test) in Kotlin/Wasm out of the box. The version of these libraries is the same as the version of the kotlin-multiplatform plugin.

Kotlin/Wasm has an experimental support for other Kotlin libraries. Read more how to enable them in your project.

Feedback

- Provide your feedback directly to developers in Kotlin Slack – get an invite and join the #webassembly channel.
- Report any problems you faced with Kotlin/Wasm on this YouTrack issue.

Kotlin Native

Kotlin/Native is a technology for compiling Kotlin code to native binaries which can run without a virtual machine. Kotlin/Native includes an LLVM-based backend for the Kotlin compiler and a native implementation of the Kotlin standard library.

Why Kotlin/Native?

Kotlin/Native is primarily designed to allow compilation for platforms on which virtual machines are not desirable or possible, such as embedded devices or iOS. It is ideal for situations when a developer needs to produce a self-contained program that does not require an additional runtime or virtual machine.

Target platforms

Kotlin/Native supports the following platforms:

- macOS
- iOS, tvOS, watchOS
- Linux
- Windows (MinGW)
- Android NDK

> To compile Apple targets, macOS, iOS, tvOS, and watchOS, you need Xcode and its command-line tools installed.

See the full list of supported targets.

Interoperability

Kotlin/Native supports two-way interoperability with native programming languages for different operating systems. The compiler creates:

- an executable for many platforms
- a static library or dynamic library with C headers for C/C++ projects
- an Apple framework for Swift and Objective-C projects

Kotlin/Native supports interoperability to use existing libraries directly from Kotlin/Native:

- static or dynamic C libraries
- C, Swift, and Objective-C frameworks

It is easy to include compiled Kotlin code in existing projects written in C, C++, Swift, Objective-C, and other languages. It is also easy to use existing native code, static or dynamic C libraries, Swift/Objective-C frameworks, graphical engines, and anything else directly from Kotlin/Native.

Kotlin/Native libraries help share Kotlin code between projects. POSIX, gzip, OpenGL, Metal, Foundation, and many other popular libraries and Apple frameworks are pre-imported and included as Kotlin/Native libraries in the compiler package.

Sharing code between platforms

Kotlin Multiplatform helps share common code across multiple platforms, including Android, iOS, JVM, web, and native. Multiplatform libraries provide the necessary APIs for common Kotlin code and allow writing shared parts of projects in Kotlin all in one place.

You can use the Get started with Kotlin Multiplatform for mobile tutorial to create applications and share business logic between iOS and Android. To share UIs among iOS, Android, desktop, and web, try Compose Multiplatform, JetBrains' declarative UI framework based on Kotlin and Jetpack Compose.

How to get started

New to Kotlin? Take a look at Getting started with Kotlin.

Recommended documentation:

- Get started with Kotlin Multiplatform
- Interoperability with C
- Interoperability with Swift/Objective-C

Recommended tutorials:

- Get started with Kotlin/Native
- Get started with Kotlin Multiplatform for mobile
- Mapping primitive data types from C
- Kotlin/Native as a dynamic Library
- Kotlin/Native as an Apple framework

Kotlin for JavaScript

Kotlin/JS provides the ability to transpile your Kotlin code, the Kotlin standard library, and any compatible dependencies to JavaScript. The current implementation of Kotlin/JS targets ES5.

The recommended way to use Kotlin/JS is via the kotlin.multiplatform Gradle plugin. It lets you easily set up and control Kotlin projects targeting JavaScript in one place. This includes essential functionality such as controlling the bundling of your application, adding JavaScript dependencies directly from npm, and more. To get an overview of the available options, check out Set up a Kotlin/JS project.

Kotlin/JS IR compiler

The Kotlin/JS IR compiler comes with a number of improvements over the old default compiler. For example, it reduces the size of generated executables via dead code elimination and provides smoother interoperability with the JavaScript ecosystem and its tooling.

> The old compiler has been deprecated since the Kotlin 1.8.0 release.

By generating TypeScript declaration files (d.ts) from Kotlin code, the IR compiler makes it easier to create "hybrid" applications that mix TypeScript and Kotlin code and to leverage code-sharing functionality using Kotlin Multiplatform.

To learn more about the available features in the Kotlin/JS IR compiler and how to try it for your project, visit the Kotlin/JS IR compiler documentation page and the migration guide.

Kotlin/JS frameworks

Modern web development benefits significantly from frameworks that simplify building web applications. Here are a few examples of popular web frameworks for Kotlin/JS written by different authors:

KVision
KVision is an object-oriented web framework that makes it possible to write applications in Kotlin/JS with ready-to-use components that can be used as building blocks for your application's user interface. You can use both reactive and imperative programming models to build your frontend, use connectors for Ktor, Spring Boot, and other frameworks to integrate it with your server-side applications, and share code using Kotlin Multiplatform.

Visit KVision site for documentation, tutorials, and examples.

For updates and discussions about the framework, join the #kvision and #javascript channels in the Kotlin Slack.

fritz2
fritz2 is a standalone framework for building reactive web user interfaces. It provides its own type-safe DSL for building and rendering HTML elements, and it makes use of Kotlin's coroutines and flows to express components and their data bindings. It provides state management, validation, routing, and more out of the box, and integrates with Kotlin Multiplatform projects.

Visit fritz2 site for documentation, tutorials, and examples.

For updates and discussions about the framework, join the #fritz2 and #javascript channels in the Kotlin Slack.

Doodle

Doodle is a vector-based UI framework for Kotlin/JS. Doodle applications use the browser's graphics capabilities to draw user interfaces instead of relying on DOM, CSS, or Javascript. By using this approach, Doodle gives you precise control over the rendering of arbitrary UI elements, vector shapes, gradients, and custom visualizations.

Visit Doodle site for documentation, tutorials, and examples.

For updates and discussions about the framework, join the #doodle and #javascript channels in the Kotlin Slack.

Join the Kotlin/JS community

You can join the #javascript channel in the official Kotlin Slack to chat with the community and the team.

Kotlin for data science

From building data pipelines to productionizing machine learning models, Kotlin can be a great choice for working with data:

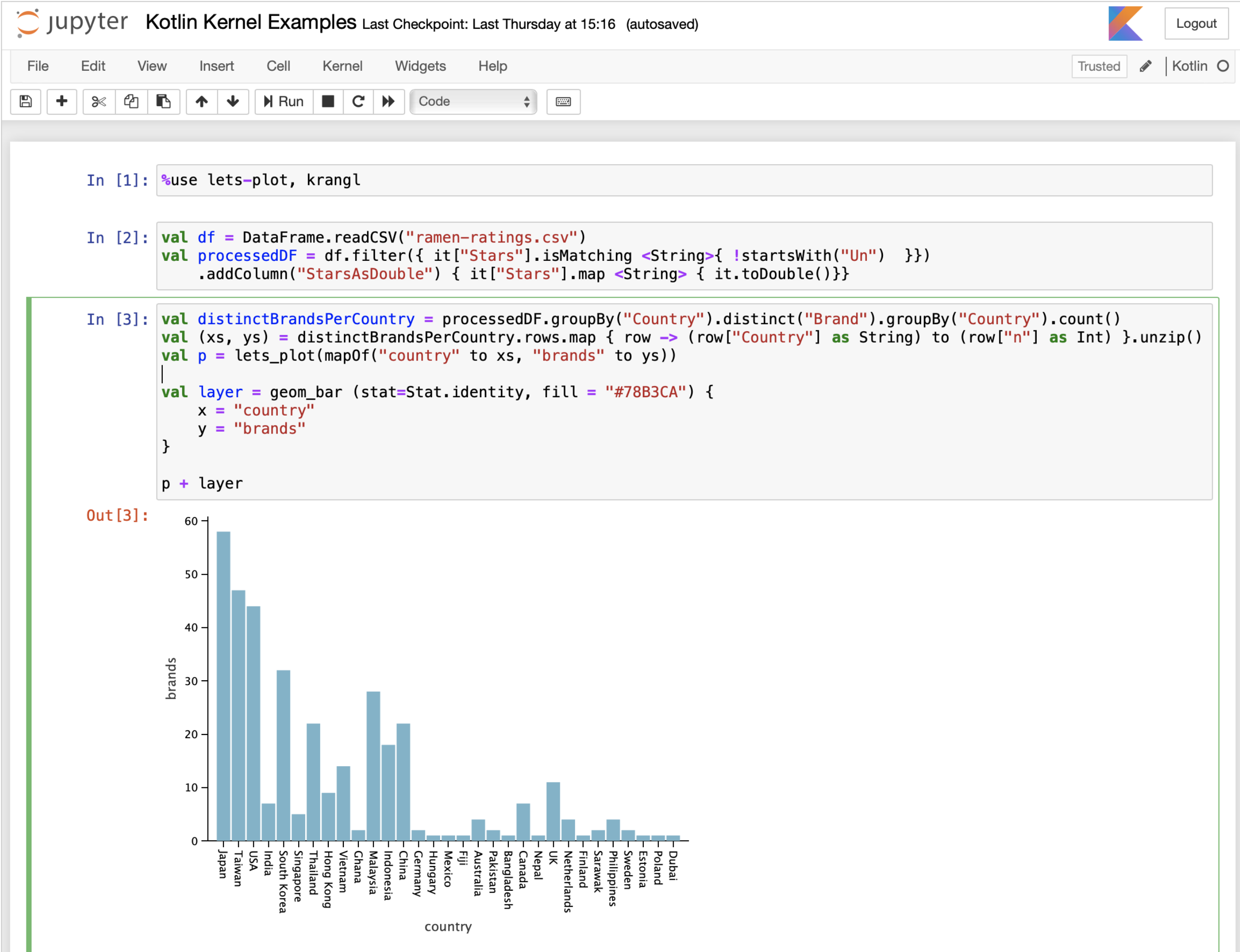
- Kotlin is concise, readable, and easy to learn.
- Static typing and null safety help create reliable, maintainable code that is easy to troubleshoot.
- Being a JVM language, Kotlin gives you great performance and an ability to leverage an entire ecosystem of tried and true Java libraries.

Interactive editors

Notebooks such as Jupyter Notebook, Datalore, and Apache Zeppelin provide convenient tools for data visualization and exploratory research. Kotlin integrates with these tools to help you explore data, share your findings with colleagues, or build up your data science and machine learning skills.

Jupyter Kotlin kernel

The Jupyter Notebook is an open-source web application that allows you to create and share documents (aka "notebooks") that can contain code, visualizations, and Markdown text. Kotlin-jupyter is an open source project that brings Kotlin support to Jupyter Notebook.

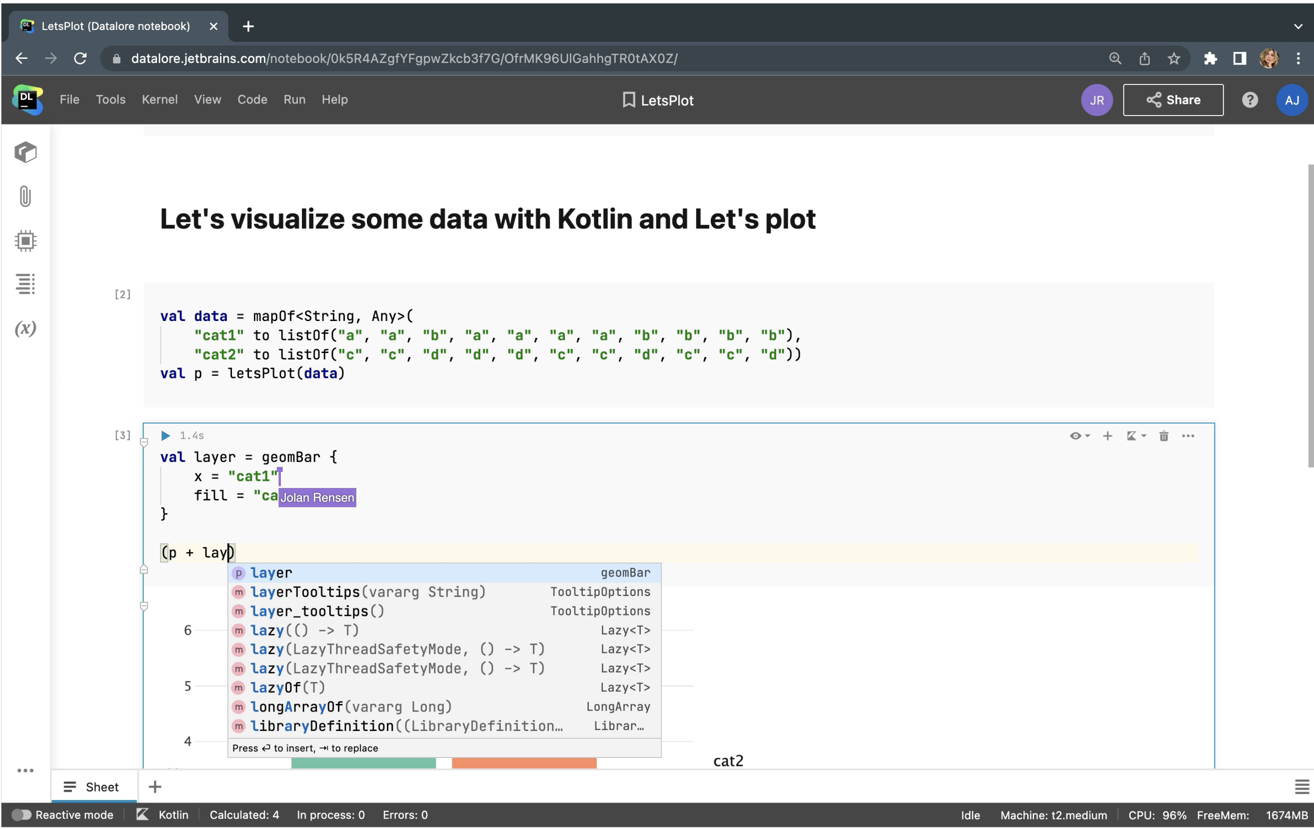


Kotlin in Jupyter notebook

Check out Kotlin kernel's GitHub repo for installation instructions, documentation, and examples.

Kotlin Notebooks in Datalore

With Datalore, you can use Kotlin in the browser straight out of the box, no installation required. You can also collaborate on Kotlin notebooks in real time, get smart coding assistance when writing code, and share results as interactive or static reports. Check out a sample report.

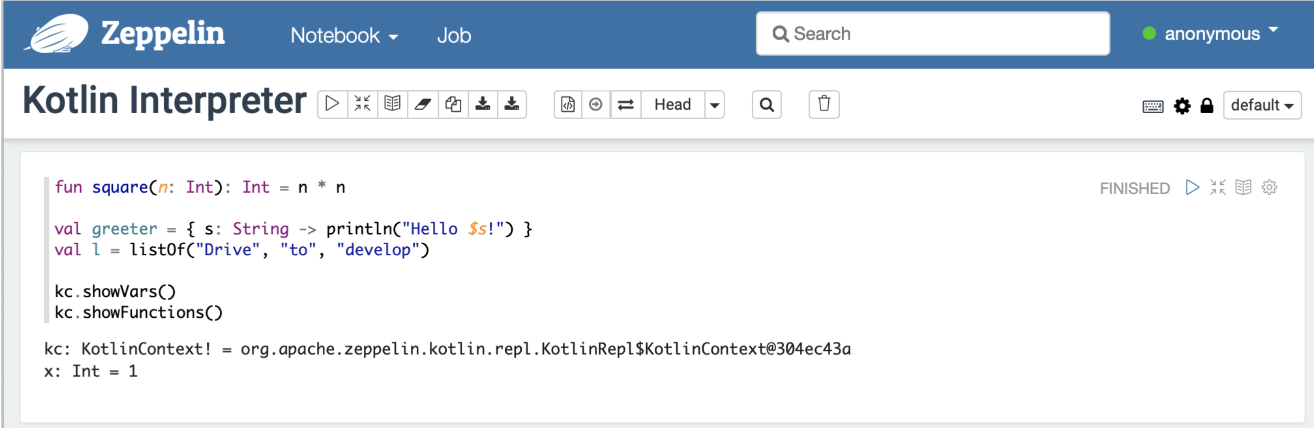


Kotlin in Datalore

Sign up and use Kotlin with a free Datalore Community account.

Zeppelin Kotlin interpreter

Apache Zeppelin is a popular web-based solution for interactive data analytics. It provides strong support for the Apache Spark cluster computing system, which is particularly useful for data engineering. Starting from version 0.9.0, Apache Zeppelin comes with bundled Kotlin interpreter.



Kotlin in Zeppelin notebook

Libraries

The ecosystem of libraries for data-related tasks created by the Kotlin community is rapidly expanding. Here are some libraries that you may find useful:

Kotlin libraries

- Multik: multidimensional arrays in Kotlin. The library provides Kotlin-idiomatic, type- and dimension-safe API for mathematical operations over multidimensional arrays. Multik offers swappable JVM and native computational engines, and a combination of the two for optimal performance.
- KotlinDL is a high-level Deep Learning API written in Kotlin and inspired by Keras. It offers simple APIs for training deep learning models from scratch, importing existing Keras models for inference, and leveraging transfer learning for tweaking existing pre-trained models to your tasks.
- Kotlin DataFrame is a library for structured data processing. It aims to reconcile Kotlin's static typing with the dynamic nature of data by utilizing both the full power of the Kotlin language and the opportunities provided by intermittent code execution in Jupyter notebooks and REPLs.
- Kotlin for Apache Spark adds a missing layer of compatibility between Kotlin and Apache Spark. It allows Kotlin developers to use familiar language features such as data classes, and lambda expressions as simple expressions in curly braces or method references.
- kotlin-statistics is a library providing extension functions for exploratory and production statistics. It supports basic numeric list/sequence/array functions (from sum to skewness), slicing operators (such as countBy, simpleRegressionBy), binning operations, discrete PDF sampling, naive bayes classifier, clustering, linear regression, and much more.
- kmath is an experimental library that was intially inspired by NumPy but evolved to more flexible abstractions. It implements mathematical operations combined in algebraic structures over Kotlin types, defines APIs for linear structures, expressions, histograms, streaming operations, provides interchangeable wrappers over existing Java and Kotlin libraries including ND4J, Commons Math, Multik, and others.
- krangl is a library inspired by R's dplyr and Python's pandas. This library provides functionality for data manipulation using a functional-style API; it also includes functions for filtering, transforming, aggregating, and reshaping tabular data.
- lets-plot is a plotting library for statistical data written in Kotlin. Lets-Plot is multiplatform and can be used not only with JVM, but also with JS and Python.
- kravis is another library for the visualization of tabular data inspired by R's ggplot.
- londogard-nlp-toolkit is a library that provides utilities when working with natural language processing such as word/subword/sentence embeddings, word-frequencies, stopwords, stemming, and much more.

Java libraries

Since Kotlin provides first-class interop with Java, you can also use Java libraries for data science in your Kotlin code. Here are some examples of such libraries:

- DeepLearning4J - a deep learning library for Java
- ND4J - an efficient matrix math library for JVM
- Dex - a Java-based data visualization tool
- Smile - a comprehensive machine learning, natural language processing, linear algebra, graph, interpolation, and visualization system. Besides Java API, Smile also provides a functional Kotlin API along with Scala and Clojure API.
  - Smile-NLP-kt - a Kotlin rewrite of the Scala implicits for the natural language processing part of Smile in the format of extension functions and interfaces.
- Apache Commons Math - a general math, statistics, and machine learning library for Java
- NM Dev - a Java mathematical library that covers all of classical mathematics.
- OptaPlanner - a solver utility for optimization planning problems
- Charts - a scientific JavaFX charting library in development
- Apache OpenNLP - a machine learning based toolkit for the processing of natural language text
- CoreNLP - a natural language processing toolkit
- Apache Mahout - a distributed framework for regression, clustering and recommendation
- Weka - a collection of machine learning algorithms for data mining tasks
- Tablesaw - a Java dataframe. It includes a visualization library based on Plot.ly

If this list doesn't cover your needs, you can find more options in the Kotlin Machine Learning Demos GitHub repository with showcases from Thomas Nield.

Kotlin for competitive programming

This tutorial is designed both for competitive programmers that did not use Kotlin before and for Kotlin developers that did not participate in any competitive programming events before. It assumes the corresponding programming skills.

Competitive programming is a mind sport where contestants write programs to solve precisely specified algorithmic problems within strict constraints. Problems can range from simple ones that can be solved by any software developer and require little code to get a correct solution, to complex ones that require knowledge of special algorithms, data structures, and a lot of practice. While not being specifically designed for competitive programming, Kotlin incidentally fits well in this domain, reducing the typical amount of boilerplate that a programmer needs to write and read while working with the code almost to the level offered by dynamically-typed scripting languages, while having tooling and performance of a statically-typed language.

See Get started with Kotlin/JVM on how to set up development environment for Kotlin. In competitive programming, a single project is usually created and each problem's solution is written in a single source file.

Simple example: Reachable Numbers problem

Let's take a look at a concrete example.

Codeforces Round 555 was held on April 26th for 3rd Division, which means it had problems fit for any developer to try. You can use this link to read the problems. The simplest problem in the set is the Problem A: Reachable Numbers. It asks to implement a straightforward algorithm described in the problem statement.

We'd start solving it by creating a Kotlin source file with an arbitrary name. A.kt will do well. First, you need to implement a function specified in the problem statement as:

Let's denote a function f(x) in such a way: we add 1 to x, then, while there is at least one trailing zero in the resulting number, we remove that zero.

Kotlin is a pragmatic and unopinionated language, supporting both imperative and function programming styles without pushing the developer towards either one. You can implement the function f in functional style, using such Kotlin features as tail recursion:

```
tailrec fun removeZeroes(x: Int): Int =
    if (x % 10 == 0) removeZeroes(x / 10) else x

fun f(x: Int) = removeZeroes(x + 1)
```

Alternatively, you can write an imperative implementation of the function f using the traditional while loop and mutable variables that are denoted in Kotlin with var:

```
fun f(x: Int): Int {
    var cur = x + 1
    while (cur % 10 == 0) cur /= 10
    return cur
}
```

Types in Kotlin are optional in many places due to pervasive use of type-inference, but every declaration still has a well-defined static type that is known at compilation.

Now, all is left is to write the main function that reads the input and implements the rest of the algorithm that the problem statement asks for — to compute the number of different integers that are produced while repeatedly applying function f to the initial number n that is given in the standard input.

By default, Kotlin runs on JVM and gives direct access to a rich and efficient collections library with general-purpose collections and data-structures like dynamically-sized arrays (ArrayList), hash-based maps and sets (HashMap/HashSet), tree-based ordered maps and sets (TreeMap/TreeSet). Using a hash-set of integers to track values that were already reached while applying function f, the straightforward imperative version of a solution to the problem can be written as shown below:

Kotlin 1.6.0 and later

```
fun main() {
    var n = readln().toInt() // read integer from the input
    val reached = HashSet<Int>() // a mutable hash set
    while (reached.add(n)) n = f(n) // iterate function f
    println(reached.size) // print answer to the output
}
```

There is no need to handle the case of misformatted input in competitive programming. An input format is always precisely specified in competitive programming, and the actual input cannot deviate from the input specification in the problem statement. That's why you can use Kotlin's readln() function. It asserts that the input string is present and throws an exception otherwise. Likewise, the String.toInt() function throws an exception if the input string is not an integer.

Earlier versions

```kotlin
fun main() {
    var n = readLine()!!.toInt() // read integer from the input
    val reached = HashSet<Int>() // a mutable hash set
    while (reached.add(n)) n = f(n) // iterate function f
    println(reached.size) // print answer to the output
}
```

Note the use of Kotlin's null-assertion operator !! after the readLine() function call. Kotlin's readLine() function is defined to return a nullable type String? and returns null on the end of the input, which explicitly forces the developer to handle the case of missing input.

There is no need to handle the case of misformatted input in competitive programming. In competitive programming, an input format is always precisely specified and the actual input cannot deviate from the input specification in the problem statement. That's what the null-assertion operator !! essentially does — it asserts that the input string is present and throws an exception otherwise. Likewise, the String.toInt().

All online competitive programming events allow the use of pre-written code, so you can define your own library of utility functions that are geared towards competitive programming to make your actual solution code somewhat easier to read and write. You would then use this code as a template for your solutions. For example, you can define the following helper functions for reading inputs in competitive programming:

Kotlin 1.6.0 and later

```kotlin
private fun readStr() = readln() // string line
private fun readInt() = readStr().toInt() // single int
// similar for other types you'd use in your solutions
```

Earlier versions

```kotlin
private fun readStr() = readLine()!! // string line
private fun readInt() = readStr().toInt() // single int
// similar for other types you'd use in your solutions
```

Note the use of private visibility modifier here. While the concept of visibility modifier is not relevant for competitive programming at all, it allows you to place multiple solution files based on the same template without getting an error for conflicting public declarations in the same package.

Functional operators example: Long Number problem

For more complicated problems, Kotlin's extensive library of functional operations on collections comes in handy to minimize the boilerplate and turn the code into a linear top-to-bottom and left-to-right fluent data transformation pipeline. For example, the Problem B: Long Number problem takes a simple greedy algorithm to implement and it can be written using this style without a single mutable variable:

Kotlin 1.6.0 and later

```kotlin
fun main() {
    // read input
    val n = readln().toInt()
    val s = readln()
    val fl = readln().split(" ").map { it.toInt() }
    // define local function f
    fun f(c: Char) = '0' + fl[c - '1']
    // greedily find first and last indices
    val i = s.indexOfFirst { c -> f(c) > c }
        .takeIf { it >= 0 } ?: s.length
    val j = s.withIndex().indexOfFirst { (j, c) -> j > i && f(c) < c }
        .takeIf { it >= 0 } ?: s.length
    // compose and write the answer
    val ans =
        s.substring(0, i) +
        s.substring(i, j).map { c -> f(c) }.joinToString("") +
        s.substring(j)
    println(ans)
}
```

Earlier versions

```kotlin
fun main() {
    // read input
    val n = readLine()!!.toInt()
    val s = readLine()!!
    val fl = readLine()!!.split(" ").map { it.toInt() }
    // define local function f
    fun f(c: Char) = '0' + fl[c - '1']
    // greedily find first and last indices
    val i = s.indexOfFirst { c -> f(c) > c }
        .takeIf { it >= 0 } ?: s.length
    val j = s.withIndex().indexOfFirst { (j, c) -> j > i && f(c) < c }
        .takeIf { it >= 0 } ?: s.length
    // compose and write the answer
    val ans =
        s.substring(0, i) +
        s.substring(i, j).map { c -> f(c) }.joinToString("") +
        s.substring(j)
    println(ans)
}
```

In this dense code, in addition to collection transformations, you can see such handy Kotlin features as local functions and the elvis operator ?: that allow to express idioms like "take the value if it is positive or else use length" with a concise and readable expressions like .takeIf { it >= 0 } ?: s.length, yet it is perfectly fine with Kotlin to create additional mutable variables and express the same code in imperative style, too.

To make reading the input in competitive programming tasks like this more concise, you can have the following list of helper input-reading functions:

Kotlin 1.6.0 and later

```kotlin
private fun readStr() = readln() // string line
private fun readInt() = readStr().toInt() // single int
private fun readStrings() = readStr().split(" ") // list of strings
private fun readInts() = readStrings().map { it.toInt() } // list of ints
```

Earlier versions

```kotlin
private fun readStr() = readLine()!! // string line
private fun readInt() = readStr().toInt() // single int
private fun readStrings() = readStr().split(" ") // list of strings
private fun readInts() = readStrings().map { it.toInt() } // list of ints
```

With these helpers, the part of code for reading input becomes simpler, closely following the input specification in the problem statement line by line:

```kotlin
// read input
val n = readInt()
val s = readStr()
val fl = readInts()
```

Note that in competitive programming it is customary to give variables shorter names than it is typical in industrial programming practice, since the code is to be written just once and not supported thereafter. However, these names are usually still mnemonic — a for arrays, i, j, and others for indices, r, and c for row and column numbers in tables, x and y for coordinates, and so on. It is easier to keep the same names for input data as it is given in the problem statement. However, more complex problems require more code which leads to using longer self-explanatory variable and function names.

More tips and tricks

Competitive programming problems often have input like this:

The first line of the input contains two integers n and k

In Kotlin this line can be concisely parsed with the following statement using destructuring declaration from a list of integers:

```kotlin
val (n, k) = readInts()
```

It might be temping to use JVM's java.util.Scanner class to parse less structured input formats. Kotlin is designed to interoperate well with JVM libraries, so that their use feels quite natural in Kotlin. However, beware that java.util.Scanner is extremely slow. So slow, in fact, that parsing 105 or more integers with it might not fit

into a typical 2 second time-limit, which a simple Kotlin's split(" ").map { it.toInt() } would handle.

Writing output in Kotlin is usually straightforward with println(...) calls and using Kotlin's string templates. However, care must be taken when output contains on order of 105 lines or more. Issuing so many println calls is too slow, since the output in Kotlin is automatically flushed after each line. A faster way to write many lines from an array or a list is using joinToString() function with "\n" as the separator, like this:

```
println(a.joinToString("\n")) // each element of array/list of a separate line
```

Learning Kotlin

Kotlin is easy to learn, especially for those who already know Java. A short introduction to the basic syntax of Kotlin for software developers can be found directly in the reference section of the website starting from basic syntax.

IDEA has built-in Java-to-Kotlin converter. It can be used by people familiar with Java to learn the corresponding Kotlin syntactic constructions, but it is not perfect, and it is still worth familiarizing yourself with Kotlin and learning the Kotlin idioms.

A great resource to study Kotlin syntax and API of the Kotlin standard library are Kotlin Koans.

What's new in Kotlin 1.9.0

Release date: July 6, 2023

The Kotlin 1.9.0 release is out and the K2 compiler for the JVM is now in Beta. Additionally, here are some of the main highlights:

- New Kotlin K2 compiler updates
- Stable replacement of the enum class values function
- Stable ..< operator for open-ended ranges
- New common function to get regex capture group by name
- New path utility to create parent directories
- Preview of Gradle configuration cache in Kotlin Multiplatform
- Changes to Android target support in Kotlin Multiplatform
- Preview of custom memory allocator in Kotlin/Native
- Library linkage in Kotlin/Native
- Size-related optimizations in Kotlin/Wasm

You can also find a short overview of the updates in this video:

Watch video online.

IDE support

The Kotlin plugins that support 1.9.0 are available for:

IDE	Supported versions
IntelliJ IDEA	2022.3.x, 2023.1.x
Android Studio	Giraffe (223), Hedgehog (231)*

*The Kotlin 1.9.0 plugin will be integrated with Android Studio Giraffe (223) and Hedgehog (231) in their upcoming releases.

The Kotlin 1.9.0 plugin will be integrated with IntelliJ IDEA 2023.2 in the upcoming releases.

> To download Kotlin artifacts and dependencies, configure Gradle settings to use the Maven Central Repository.

New Kotlin K2 compiler updates

The Kotlin team at JetBrains continues to stabilize the K2 compiler and the 1.9.0 release introduces further advancements. The K2 compiler for the JVM is now in Beta.

There's now also basic support for Kotlin/Native and multiplatform projects.

Compatibility of the kapt compiler plugin with the K2 compiler

You can use the kapt plugin in your project along with the K2 compiler, but with some restrictions. Despite setting languageVersion to 2.0, the kapt compiler plugin still utilizes the old compiler.

If you execute the kapt compiler plugin within a project where languageVersion is set to 2.0, kapt will automatically switch to 1.9 and disable specific version compatibility checks. This behavior is equivalent to including the following command arguments:

- -Xskip-metadata-version-check
- -Xskip-prerelease-check
- -Xallow-unstable-dependencies

These checks are exclusively disabled for kapt tasks. All other compilation tasks will continue to utilize the new K2 compiler.

If you encounter any issues when using kapt with the K2 compiler, please report them to our issue tracker.

Try the K2 compiler in your project

Starting with 1.9.0 until the release of Kotlin 2.0, you can easily test the K2 compiler with the kotlin.experimental.tryK2=true Gradle property. You can also run the following command:

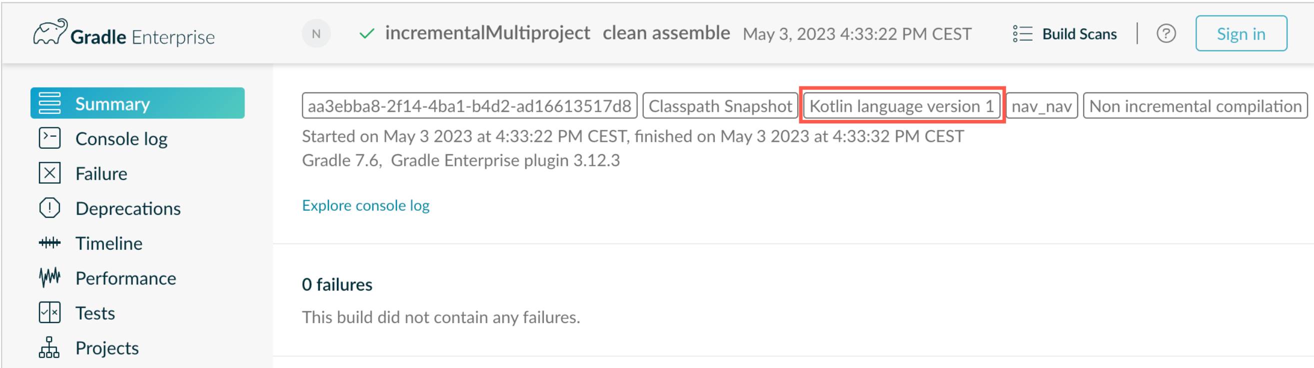
```
./gradlew assemble -Pkotlin.experimental.tryK2=true
```

This Gradle property automatically sets the language version to 2.0 and updates the build report with the number of Kotlin tasks compiled using the K2 compiler compared to the current compiler:

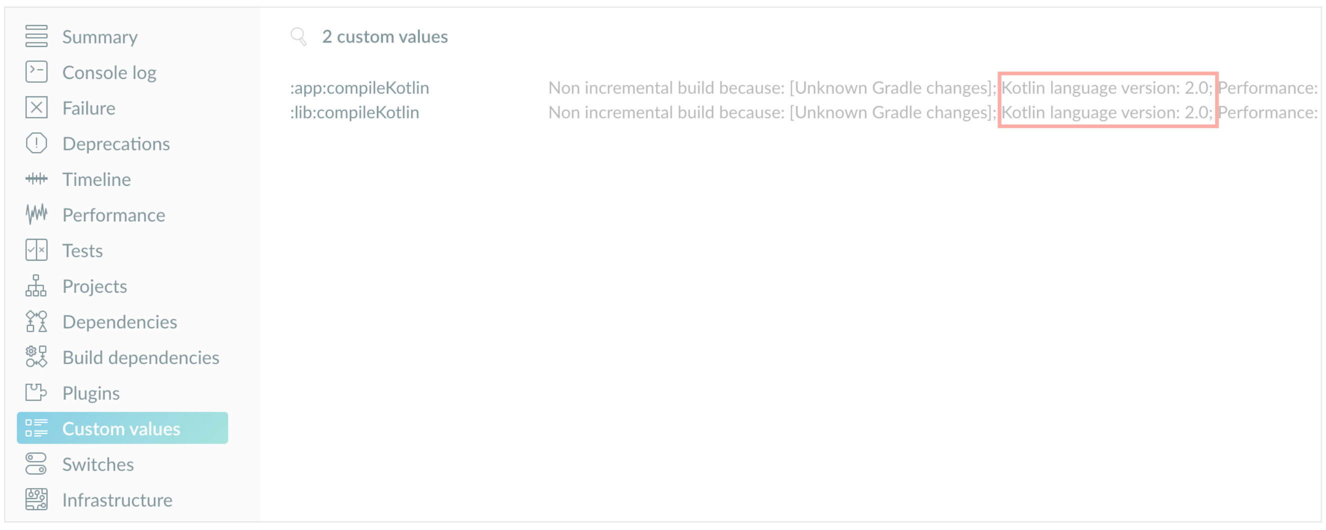
```
##### 'kotlin.experimental.tryK2' results (Kotlin/Native not checked) #####
:lib:compileKotlin: 2.0 language version
:app:compileKotlin: 2.0 language version
##### 100% (2/2) tasks have been compiled with Kotlin 2.0 #####
```

Gradle build reports

Gradle build reports now show whether the current or the K2 compiler was used to compile the code. In Kotlin 1.9.0, you can see this information in your Gradle build scans:



Gradle build scan - K1



Gradle build scan - K2

You can also find the Kotlin version used in the project right in the build report:

```
Task info:
  Kotlin language version: 1.9
```

> If you use Gradle 8.0, you might come across some problems with build reports, especially when Gradle configuration caching is enabled. This is a known issue, fixed in Gradle 8.1 and later.

Current K2 compiler limitations

Enabling K2 in your Gradle project comes with certain limitations that can affect projects using Gradle versions below 8.3 in the following cases:

- Compilation of source code from buildSrc.
- Compilation of Gradle plugins in included builds.
- Compilation of other Gradle plugins if they are used in projects with Gradle versions below 8.3.
- Building Gradle plugin dependencies.

If you encounter any of the problems mentioned above, you can take the following steps to address them:

- Set the language version for buildSrc, any Gradle plugins, and their dependencies:

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
    }
}
```

- Update the Gradle version in your project to 8.3 when it becomes available.

Leave your feedback on the new K2 compiler

We'd appreciate any feedback you may have!

- Provide your feedback directly to K2 developers Kotlin's Slack – get an invite and join the #k2-early-adopters channel.
- Report any problems you've faced with the new K2 compiler on our issue tracker.
- Enable the Send usage statistics option to allow JetBrains to collect anonymous data about K2 usage.

Language

In Kotlin 1.9.0, we're stabilizing some new language features that were introduced earlier:

- Replacement of the enum class values function
- Data object symmetry with data classes
- Support for secondary constructors with bodies in inline value classes

Stable replacement of the enum class values function

In 1.8.20, the entries property for enum classes was introduced as an Experimental feature. The entries property is a modern and performant replacement for the synthetic values() function. In 1.9.0, the entries property is Stable.

> The values() function is still supported, but we recommend that you use the entries property instead.

```
enum class Color(val colorName: String, val rgb: String) {
    RED("Red", "#FF0000"),
    ORANGE("Orange", "#FF7F00"),
    YELLOW("Yellow", "#FFFF00")
}

fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb == rgb }
```

For more information about the entries property for enum classes, see What's new in Kotlin 1.8.20.

Stable data objects for symmetry with data classes

Data object declarations that were introduced in Kotlin 1.8.20 are now Stable. This includes the functions added for symmetry with data classes: toString(), equals(), and hashCode().

This feature is particularly useful with sealed hierarchies (like a sealed class or sealed interface hierarchy), because data object declarations can be used conveniently alongside data class declarations. In this example, declaring EndOfFile as a data object instead of a plain object means that it automatically has a toString() function without the need to override it manually. This maintains symmetry with the accompanying data class definitions.

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // Number(number=7)
```

```
    println(EndOfFile) // EndOfFile
}
```

For more information, see What’s new in Kotlin 1.8.20.

Support for secondary constructors with bodies in inline value classes

Starting with Kotlin 1.9.0, the use of secondary constructors with bodies in inline value classes is available by default:

```
@JvmInline
value class Person(private val fullName: String) {
    // Allowed since Kotlin 1.4.30:
    init {
        check(fullName.isNotBlank()) {
            "Full name shouldn't be empty"
        }
    }
    // Allowed by default since Kotlin 1.9.0:
    constructor(name: String, lastName: String) : this("$name $lastName") {
        check(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }
}
```

Previously, Kotlin allowed using only public primary constructors in inline classes. As a result, it was impossible to encapsulate underlying values or create an inline class that would represent some constrained values.

As Kotlin developed, these issues were fixed. Kotlin 1.4.30 lifted restrictions on init blocks and then Kotlin 1.8.20 came with a preview of secondary constructors with bodies. They are now available by default. Learn more about the development of Kotlin inline classes in this KEEP.

Kotlin/JVM

Starting with version 1.9.0, the compiler can generate classes with a bytecode version corresponding to JVM 20. In addition, the deprecation of the JvmDefault annotation and legacy -Xjvm-default modes continues.

Deprecation of JvmDefault annotation and legacy -Xjvm-default modes

Starting from Kotlin 1.5, the usage of the JvmDefault annotation has been deprecated in favor of the newer -Xjvm-default modes: all and all-compatibility. With the introduction of JvmDefaultWithoutCompatibility in Kotlin 1.4 and JvmDefaultWithCompatibility in Kotlin 1.6, these modes offer comprehensive control over the generation of DefaultImpls classes, ensuring seamless compatibility with older Kotlin code.

Consequently in Kotlin 1.9.0, the JvmDefault annotation no longer holds any significance and has been marked as deprecated, resulting in an error. It will eventually be removed from Kotlin.

Kotlin/Native

Among other improvements, this release brings further advancements to the Kotlin/Native memory manager that should enhance its robustness and performance:

- Preview of custom memory allocator
- Objective-C or Swift object deallocation hook on the main thread
- No object initialization when accessing constant values in Kotlin/Native
- Ability to configure standalone mode for iOS simulator tests
- Library linkage in Kotlin/Native

Preview of custom memory allocator

Kotlin 1.9.0 introduces the preview of a custom memory allocator. Its allocation system improves the runtime performance of the Kotlin/Native memory manager.

The current object allocation system in Kotlin/Native uses a general-purpose allocator that doesn't have the functionality for efficient garbage collection. To compensate, it maintains thread-local linked lists of all allocated objects before the garbage collector (GC) merges them into a single list, which can be iterated

during sweeping. This approach comes with several performance downsides:

- The sweeping order lacks memory locality and often results in scattered memory access patterns, leading to potential performance issues.
- Linked lists require additional memory for each object, increasing memory usage, particularly when dealing with many small objects.
- The single list of allocated objects makes it challenging to parallelize sweeping, which can cause memory usage problems when mutator threads allocate objects faster than the GC thread can collect them.

To address these issues, Kotlin 1.9.0 introduces a preview of the custom allocator. It divides system memory into pages, allowing independent sweeping in consecutive order. Each allocation becomes a memory block within a page, and the page keeps track of block sizes. Different page types are optimized for various allocation sizes. The consecutive arrangement of memory blocks ensures efficient iteration through all allocated blocks.

When a thread allocates memory, it searches for a suitable page based on the allocation size. Threads maintain a set of pages for different size categories. Typically, the current page for a given size can accommodate the allocation. If not, the thread requests a different page from the shared allocation space. This page may already be available, require sweeping, or should be created first.

The new allocator allows having multiple independent allocation spaces simultaneously, which will allow the Kotlin team to experiment with different page layouts to improve performance even further.

For more information on the design of the new allocator, see this README.

How to enable

Add the -Xallocator=custom compiler option:

```
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                freeCompilerArgs.add("-Xallocator=custom")
            }
        }
    }
}
```

Leave feedback

We would appreciate your feedback in YouTrack to improve the custom allocator.

Objective-C or Swift object deallocation hook on the main thread

Starting with Kotlin 1.9.0, the Objective-C or Swift object deallocation hook is called on the main thread if the object is passed to Kotlin on the main thread. The way the Kotlin/Native memory manager previously handled references to Objective-C objects could lead to memory leaks. We believe the new behavior should improve the robustness of the memory manager.

Consider an Objective-C object that is referenced in Kotlin code, for example, when passed as an argument, returned by a function, or retrieved from a collection. In this case, Kotlin creates its own object that holds the reference to the Objective-C object. When the Kotlin object gets deallocated, the Kotlin/Native runtime calls the objc_release function that releases that Objective-C reference.

Previously, the Kotlin/Native memory manager ran objc_release on a special GC thread. If it's the last object reference, the object gets deallocated. Issues could come up when Objective-C objects have custom deallocation hooks like the dealloc method in Objective-C or the deinit block in Swift, and these hooks expect to be called on a specific thread.

Since hooks for objects on the main thread usually expect to be called there, Kotlin/Native runtime now calls objc_release on the main thread as well. It should cover the cases when the Objective-C object was passed to Kotlin on the main thread, creating a Kotlin peer object there. This only works if the main dispatch queue is processed, which is the case for regular UI applications. When it's not the main queue or the object was passed to Kotlin on a thread other than main, the objc_release is called on a special GC thread as before.

How to opt out

In case you face issues, you can disable this behavior in your gradle.properties file with the following option:

```
kotlin.native.binary.objcDisposeOnMain=false
```

Don't hesitate to report such cases to our issue tracker.

No object initialization when accessing constant values in Kotlin/Native

Starting with Kotlin 1.9.0, the Kotlin/Native backend doesn't initialize objects when accessing const val fields:

```
object MyObject {
    init {
        println("side effect!")
    }

    const val y = 1
}

fun main() {
    println(MyObject.y) // No initialization at first
    val x = MyObject    // Initialization occurs
    println(x.y)
}
```

The behavior is now unified with Kotlin/JVM, where the implementation is consistent with Java and objects are never initialized in this case. You can also expect some performance improvements in your Kotlin/Native projects thanks to this change.

Ability to configure standalone mode for iOS simulator tests in Kotlin/Native

By default, when running iOS simulator tests for Kotlin/Native, the --standalone flag is used to avoid manual simulator booting and shutdown. In 1.9.0, you can now configure whether this flag is used in a Gradle task via the standalone property. By default, the --standalone flag is used so standalone mode is enabled.

Here is an example of how to disable standalone mode in your build.gradle.kts file:

```
tasks.withType<org.jetbrains.kotlin.gradle.targets.native.tasks.KotlinNativeSimulatorTest>().configureEach {
    standalone.set(false)
}
```

> If you disable standalone mode, you must boot the simulator manually. To boot your simulator from CLI, you can use the following command:
>
> ```
> /usr/bin/xcrun simctl boot <DeviceId>
> ```

Library linkage in Kotlin/Native

Starting with Kotlin 1.9.0, the Kotlin/Native compiler treats linkage issues in Kotlin libraries the same way as Kotlin/JVM. You might face such issues if the author of one third-party Kotlin library makes an incompatible change in experimental APIs that another third-party Kotlin library consumes.

Now builds don't fail during compilation in case of linkage issues between third-party Kotlin libraries. Instead, you'll only encounter these errors in run time, exactly as on the JVM.

The Kotlin/Native compiler reports warnings every time it detects issues with library linkage. You can find such warnings in your compilation logs, for example:

```
No function found for symbol 'org.samples/MyRemovedClass.doSomething|3657632771909858561[0]'

Can not get instance of singleton 'MyEnumClass.REMOVED_ENTRY': No enum entry found for symbol
'org.samples/MyEnumClass.REMOVED_ENTRY|null[0]'

Function 'getMyRemovedClass' can not be called: Function uses unlinked class symbol 'org.samples/MyRemovedClass|null[0]'
```

You can further configure or even disable this behavior in your projects:

- If you don't want to see these warnings in your compilation logs, suppress them with the -Xpartial-linkage-loglevel=INFO compiler option.
- It's also possible to raise the severity of reported warnings to compilation errors with -Xpartial-linkage-loglevel=ERROR. In this case, the compilation fails and you'll see all the errors in the compilation log. Use this option to examine the linkage issues more closely.
- If you face unexpected problems with this feature, you can always opt out with the -Xpartial-linkage=disable compiler option. Don't hesitate to report such cases to our issue tracker.

```kotlin
// An example of passing compiler options via Gradle build file.
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {

                // To suppress linkage warnings:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=INFO")

                // To raise linkage warnings to errors:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=ERROR")

                // To disable the feature completely:
                freeCompilerArgs.add("-Xpartial-linkage=disable")
            }
        }
    }
}
```

Compiler option for C interop implicit integer conversions

As we continue to work towards the stabilization of Kotlin/Native, we have introduced a compiler option for C interop that allows you to use implicit integer conversions. Previously it wasn't necessary to configure a compiler option to use implicit integer conversions. After careful consideration, we've introduced this compiler option to prevent unintentional use as this feature still has room for improvement and our aim is to have an API of the highest quality.

In this code sample an implicit integer conversion allows options = 0 even though options has unsigned type UInt and 0 is signed.

```kotlin
val today = NSDate()
val tomorrow = NSCalendar.currentCalendar.dateByAddingUnit(
    unit = NSCalendarUnitDay,
    value = 1,
    toDate = today,
    options = 0
)
```

To use implicit conversions with native interop libraries, use the -XXLanguage:+ImplicitSignedToUnsignedIntegerConversion compiler option.

You can configure this in your Gradle build.gradle.kts file:

```kotlin
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile>().configureEach {
    compilerOptions.freeCompilerArgs.addAll(
        "-XXLanguage:+ImplicitSignedToUnsignedIntegerConversion"
    )
}
```

Kotlin Multiplatform

Kotlin Multiplatform has received some notable updates in 1.9.0 designed to improve your developer experience:

- Changes to Android target support
- New Android source set layout enabled by default
- Preview of Gradle configuration cache in multiplatform projects

Changes to Android target support

We continue our efforts to stabilize Kotlin Multiplatform. An essential step is to provide first-class support for the Android target. We're excited to announce that in the future, the Android team from Google will provide its own Gradle plugin to support Android in Kotlin Multiplatform.

To open the way for this new solution from Google, we're renaming the android block in the current Kotlin DSL in 1.9.0. Please change all the occurrences of the android block to androidTarget in your build scripts. This is a temporary change that is necessary to free the android name for the upcoming DSL from Google.

The Google plugin will be the preferred way of working with Android in multiplatform projects. When it's ready, we'll provide the necessary migration instructions so that you'll be able to use the short android name as before.

New Android source set layout enabled by default

Starting with Kotlin 1.9.0, the new Android source set layout is the default. It replaced the previous naming schema for directories, which was confusing in multiple ways. The new layout has a number of advantages:

- Simplified type semantics. The new Android source layout provides clear and consistent naming conventions that help to distinguish between different types of source sets.
- Improved source directory layout. With the new layout, the SourceDirectories arrangement becomes more coherent, making it easier to organize code and locate source files.
- Clear naming schema for Gradle configurations. The schema is now more consistent and predictable in both KotlinSourceSets and AndroidSourceSets.

The new layout requires the Android Gradle plugin version 7.0 or later and is supported in Android Studio 2022.3 and later. See our migration guide to make the necessary changes in your build.gradle(.kts) file.

Preview of Gradle configuration cache

Kotlin 1.9.0 comes with support for the Gradle configuration cache in multiplatform libraries. If you're a library author, you can already benefit from the improved build performance.

Gradle configuration cache speeds up the build process by reusing the results of the configuration phase for subsequent builds. The feature has become Stable since Gradle 8.1. To enable it, follow the instructions in the Gradle documentation.

> The Kotlin Multiplatform plugin still doesn't support Gradle configuration cache with Xcode integration tasks or the Kotlin CocoaPods Gradle plugin. We expect to add this feature in future Kotlin releases.

Kotlin/Wasm

The Kotlin team continues to experiment with the new Kotlin/Wasm target. This release introduces several performance and size-related optimizations, along with updates in JavaScript interop.

Size-related optimizations

Kotlin 1.9.0 introduces significant size improvements for WebAssembly (Wasm) projects. Comparing two "Hello World" projects, the code footprint for Wasm in Kotlin 1.9.0 is now over 10 times smaller than in Kotlin 1.8.20.

These size optimizations result in more efficient resource utilization and improved performance when targeting Wasm platforms with Kotlin code.

Updates in JavaScript interop

This Kotlin update introduces changes to the interoperability between Kotlin and JavaScript for Kotlin/Wasm. As Kotlin/Wasm is an Experimental feature, certain limitations apply to its interoperability.

Restriction of Dynamic types

Starting with version 1.9.0, Kotlin no longer supports the use of Dynamic types in Kotlin/Wasm. This is now deprecated in favor of the new universal JsAny type, which facilitates JavaScript interoperability.

For more details, see the Kotlin/Wasm interoperability with JavaScript documentation.

Restriction of non-external types

Kotlin/Wasm supports conversions for specific Kotlin static types when passing values to and from JavaScript. These supported types include:

- Primitives, such as signed numbers, Boolean, Char.
- String.
- Function types.

Other types were passed without conversion as opaque references, leading to inconsistencies between JavaScript and Kotlin subtyping.

To address this, Kotlin restricts JavaScript interop to a well-supported set of types. Starting from Kotlin 1.9.0, only external, primitive, string, and function types are supported in Kotlin/Wasm JavaScript interop. Furthermore, a separate explicit type called JsReference has been introduced to represent handles to Kotlin/Wasm

objects that can be used in JavaScript interop.

For more details, refer to the Kotlin/Wasm interoperability with JavaScript documentation.

Kotlin/Wasm in Kotlin Playground

Kotlin Playground supports the Kotlin/Wasm target. You can write, run, and share your Kotlin code that targets the Kotlin/Wasm. Check it out!

> Using Kotlin/Wasm requires enabling experimental features in your browser.
>
> Learn more about how to enable these features.

```kotlin
import kotlin.time.*
import kotlin.time.measureTime

fun main() {
    println("Hello from Kotlin/Wasm!")
    computeAck(3, 10)
}

tailrec fun ack(m: Int, n: Int): Int = when {
    m == 0 -> n + 1
    n == 0 -> ack(m - 1, 1)
    else -> ack(m - 1, ack(m, n - 1))
}

fun computeAck(m: Int, n: Int) {
    var res = 0
    val t = measureTime {
        res = ack(m, n)
    }
    println()
    println("ack($m, $n) = ${res}")
    println("duration: ${t.inWholeNanoseconds / 1e6} ms")
}
```

Kotlin/JS

This release introduces updates for Kotlin/JS, including the deprecation of the Kotlin/JS Gradle plugin and Experimental support for ES6:

- Deprecation of the Kotlin/JS Gradle plugin
- Deprecation of external enum
- Experimental support for ES6 classes and modules
- Changed default destination of JS production distribution
- Extract org.w3c declarations from stdlib-js

> Starting from version 1.9.0, partial library linkage is also enabled for Kotlin/JS.

Deprecation of the Kotlin/JS Gradle plugin

Starting with Kotlin 1.9.0, the kotlin-js Gradle plugin is deprecated. We encourage you to use the kotlin-multiplatform Gradle plugin with the js() target instead.

The functionality of the Kotlin/JS Gradle plugin essentially duplicated the kotlin-multiplatform plugin and shared the same implementation under the hood. This overlap created confusion and increased maintenance load on the Kotlin team.

Refer to our Compatibility guide for Kotlin Multiplatform for migration instructions. If you find any issues that aren't covered in the guide, please report them to our issue tracker.

Deprecation of external enum

In Kotlin 1.9.0, the use of external enums will be deprecated due to issues with static enum members like entries, that can't exist outside Kotlin. We recommend

using an external sealed class with object subclasses instead:

```
// Before
external enum class ExternalEnum { A, B }

// After
external sealed class ExternalEnum {
    object A: ExternalEnum
    object B: ExternalEnum
}
```

By switching to an external sealed class with object subclasses, you can achieve similar functionality to external enums while avoiding the problems associated with default methods.

Starting from Kotlin 1.9.0, the use of external enums will be marked as deprecated. We encourage you to update your code to utilize the suggested external sealed class implementation for compatibility and future maintenance.

Experimental support for ES6 classes and modules

This release introduces Experimental support for ES6 modules and generation of ES6 classes:

- Modules offer a way to simplify your codebase and improve maintainability.
- Classes allow you to incorporate object-oriented programming (OOP) principles, resulting in cleaner and more intuitive code.

To enable these features, update your build.gradle.kts file accordingly:

```
// build.gradle.kts
kotlin {
    js(IR) {
        useEsModules() // Enables ES6 modules
        browser()
        }
    }

// Enables ES6 classes generation
tasks.withType<KotlinJsCompile>().configureEach {
    kotlinOptions {
        useEsClasses = true
    }
}
```

Learn more about ECMAScript 2015 (ES6) in the official documentation.

Changed default destination of JS production distribution

Prior to Kotlin 1.9.0, the distribution target directory was build/distributions. However, this is a common directory for Gradle archives. To resolve this issue, we've changed the default distribution target directory in Kotlin 1.9.0 to: build/dist/<targetName>/<binaryName>.

For example, productionExecutable was in build/distributions. In Kotlin 1.9.0, it's in build/dist/js/productionExecutable.

> If you have a pipeline in place that uses the results of these builds, make sure to update the directory.

Extract org.w3c declarations from stdlib-js

Since Kotlin 1.9.0, the stdlib-js no longer includes org.w3c declarations. Instead, these declarations have been moved to a separate Gradle dependency. When you add the Kotlin Multiplatform Gradle plugin to your build.gradle.kts file, these declarations will be automatically included in your project, similar to the standard library.

There is no need for any manual action or migration. The necessary adjustments will be handled automatically.

Gradle

Kotlin 1.9.0 comes with new Gradle compiler options and a lot more:

- Removed classpath property

- New Gradle compiler options
- Project-level compiler options for Kotlin/JVM
- Compiler option for Kotlin/Native module name
- Separate compiler plugins for official Kotlin libraries
- Incremented the minimum supported version
- kapt doesn't cause eager task creation
- Programmatic configuration of the JVM target validation mode

Removed classpath property

In Kotlin 1.7.0, we announced the start of a deprecation cycle for the KotlinCompile task's property: classpath. The deprecation level was raised to ERROR in Kotlin 1.8.0. In this release, we finally remove the classpath property. All compile tasks should now use the libraries input for a list of libraries required for compilation.

New compiler options

The Kotlin Gradle plugin now provides new properties for opt-ins and the compiler's progressive mode.

- To opt in to new APIs, you can now use the optIn property and pass a list of strings like: optIn.set(listOf(a, b, c)).
- To enable the progressive mode, use progressiveMode.set(true).

Project-level compiler options for Kotlin/JVM

Starting with Kotlin 1.9.0, there is a new compilerOptions block available inside the kotlin configuration block:

```
kotlin {
    compilerOptions {
        jvmTarget.set(JVM.Target_11)
    }
}
```

It makes configuring compiler options much easier. However, note some important details:

- This configuration only works on the project level.
- For the Android plugin, this block configures the same object as:

```
android {
    kotlinOptions {}
}
```

- The android.kotlinOptions and kotlin.compilerOptions configuration blocks override each other. The last (lowest) block in the build file always takes effect.
- If moduleName is configured on the project level, its value could be changed when passed to the compiler. It's not the case for the main compilation, but for other types, for example, test sources, the Kotlin Gradle plugin will add the _test suffix.
- The configuration inside the tasks.withType<KotlinJvmCompile>().configureEach {} (or tasks.named<KotlinJvmCompile>("compileKotlin") { }) overrides both kotlin.compilerOptions and android.kotlinOptions.

Compiler option for Kotlin/Native module name

The Kotlin/Native module-name compiler option is now easily available in the Kotlin Gradle plugin.

This option specifies a name for the compilation module and can also be used for adding a name prefix for the declarations exported to Objective-C.

You can now set the module name directly in the compilerOptions block of your Gradle build files:

Kotlin

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile>("compileKotlinLinuxX64") {
    compilerOptions {
```

```
        moduleName.set("my-module-name")
    }
}
```

Groovy

```
tasks.named("compileKotlinLinuxX64", org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile.class) {
    compilerOptions {
        moduleName = "my-module-name"
    }
}
```

Separate compiler plugins for official Kotlin libraries

Kotlin 1.9.0 introduces separate compiler plugins for its official libraries. Previously, compiler plugins were embedded into their corresponding Gradle plugins. This could cause compatibility issues in case the compiler plugin was compiled against a Kotlin version higher than the Gradle build's Kotlin runtime version.

Now compiler plugins are added as separate dependencies, so you won't face compatibility issues with older Gradle versions anymore. Another major advantage of the new approach is that new compiler plugins can be used with other build systems like Bazel.

Here's the list of new compiler plugins we're now publishing to Maven Central:

- kotlin-atomicfu-compiler-plugin
- kotlin-allopen-compiler-plugin
- kotlin-lombok-compiler-plugin
- kotlin-noarg-compiler-plugin
- kotlin-sam-with-receiver-compiler-plugin
- kotlinx-serialization-compiler-plugin

Every plugin has its -embeddable counterpart, for example, kotlin-allopen-compiler-plugin-embeddable is designed for working with the kotlin-compiler-embeddable artifact, the default option for scripting artifacts.

Gradle adds these plugins as compiler arguments. You don't need to make any changes to your existing projects.

Incremented minimum supported version

Starting with Kotlin 1.9.0, the minimum supported Android Gradle plugin version is 4.2.2.

See the Kotlin Gradle plugin compatibility with available Gradle versions in our documentation.

kapt doesn't cause eager task creation in Gradle

Prior to 1.9.0, the kapt compiler plugin caused eager task creation by requesting the configured instance of the Kotlin compilation task. This behavior has been fixed in Kotlin 1.9.0. If you use the default configuration for your build.gradle.kts file then your setup is not affected by this change.

> If you use a custom configuration, your setup will be adversely affected. For example, if you have modified the KotlinJvmCompile task using Gradle's tasks API, you must similarly modify the KaptGenerateStubs task in your build script.
>
> For example, if your script has the following configuration for KotlinJvmCompile task:
>
> ```
> tasks.named<KotlinJvmCompile>("compileKotlin") { // Your custom configuration }
> ```
>
> Then you need to make sure that the same modification is included as part of the KaptGenerateStubs task:
>
> ```
> tasks.named<KaptGenerateStubs>("kaptGenerateStubs") { // Your custom configuration }
> ```

For more information, see our YouTrack ticket.

Programmatic configuration of the JVM target validation mode

Before Kotlin 1.9.0, there was only one way to adjust the detection of JVM target incompatibility between Kotlin and Java. You had to set kotlin.jvm.target.validation.mode=ERROR in your gradle.properties for the whole project.

You can now also configure it on the task level in your build.gradle.kts file:

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile>("compileKotlin") {
    jvmTargetValidationMode.set(org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidationMode.WARNING)
}
```

Standard library

Kotlin 1.9.0 has some great improvements for the standard library:

- The ..< operator and time API are Stable.
- The Kotlin/Native standard library has been thoroughly reviewed and updated
- The @Volatile annotation can be used on more platforms
- There's a common function to get a regex capture group by name
- The HexFormat class is introduced to format and parse hexadecimals

Stable ..< operator for open-ended ranges

The new ..< operator for open-ended ranges that was introduced in Kotlin 1.7.20 and became Stable in 1.8.0. In 1.9.0, the standard library API for working with open-ended ranges is also Stable.

Our research shows that the new ..< operator makes it easier to understand when an open-ended range is declared. If you use the until infix function, it's easy to make the mistake of assuming that the upper bound is included.

Here is an example using the until function:

```
fun main() {
    for (number in 2 until 10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 2 4 6 8
}
```

And here is an example using the new ..< operator:

```
fun main() {
    for (number in 2..<10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 2 4 6 8
}
```

From IntelliJ IDEA version 2023.1.1, there is a new code inspection that highlights when you can use the ..< operator.

For more information about what you can do with this operator, see What's new in Kotlin 1.7.20.

Stable time API

Since 1.3.50, we have previewed a new time measurement API. The duration part of the API became Stable in 1.6.0. In 1.9.0, the remaining time measurement API is Stable.

The old time API provided measureTimeMillis and measureNanoTime functions that aren't intuitive to use. Although it is clear that they both measure time in

different units, it isn't clear that measureTimeMillisuses a wall clock to measure time, whereas measureNanoTime uses a monotonic time source. The new time API resolves this and other issues to make the API more user friendly.

With the new time API, you can easily:

- Measure the time taken to execute some code using a monotonic time source with your desired time unit.
- Mark a moment in time.
- Compare and subtract two moments in time.
- Check how much time has passed since a specific moment in time.
- Check whether the current time has passed a specific moment in time.

Measure code execution time

To measure the time taken to execute a block of code, use the measureTime inline function.

To measure the time taken to execute a block of code and return the result of the block of code, use the measureTimedValue inline function.

By default, both functions use a monotonic time source. However, if you want to use an elapsed real-time source, you can. For example, on Android the default time source System.nanoTime() only counts time while the device is active. It loses track of time when the device enters deep sleep. To keep track of time while the device is in deep sleep, you can create a time source that uses SystemClock.elapsedRealtimeNanos() instead:

```
object RealtimeMonotonicTimeSource : AbstractLongTimeSource(DurationUnit.NANOSECONDS) {
    override fun read(): Long = SystemClock.elapsedRealtimeNanos()
}
```

Mark and measure differences in time

To mark a specific moment in time, use the TimeSource interface and the markNow() function to create a TimeMark. To measure differences between TimeMarks from the same time source, use the subtraction operator (-):

```
fun main() {
//sampleStart
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    Thread.sleep(500) // Sleep 0.5 seconds.
    val mark2 = timeSource.markNow()

    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        println("Measurement 1.${n + 1}: elapsed1=$elapsed1, elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    // It's also possible to compare time marks with each other.
    println(mark2 > mark1) // This is true, as mark2 was captured later than mark1.
//sampleEnd
}
```

To check if a deadline has passed or a timeout has been reached, use the hasPassedNow() and hasNotPassedNow() extension functions:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    val fiveSeconds: Duration = 5.seconds
    val mark2 = mark1 + fiveSeconds

    // It hasn't been 5 seconds yet
    println(mark2.hasPassedNow())
    // false

    // Wait six seconds
    Thread.sleep(6000)
    println(mark2.hasPassedNow())
    // true
```

```
//sampleEnd
}
```

The Kotlin/Native standard library's journey towards stabilization

As our standard library for Kotlin/Native continues to grow, we decided that it was time for a complete review to ensure that it meets our high standards. As part of this, we carefully reviewed every existing public signature. For each signature, we considered whether it:

- Has a unique purpose.
- Is consistent with other Kotlin APIs.
- Has similar behavior to its counterpart for the JVM.
- Is future-proof.

Based on these considerations, we made one of the following decisions:

- Made it Stable.
- Made it Experimental.
- Marked it as private.
- Modified the behavior.
- Moved it to a different location.
- Deprecated it.
- Marked it as obsolete.

> If an existing signature has been:
>
> - Moved to another package, the signature still exists in the original package but it's now deprecated with deprecation level: WARNING. IntelliJ IDEA will automatically suggest replacements upon code inspection.
> - Deprecated, it's been deprecated with deprecation level: WARNING.
> - Marked as obsolete, you can keep using it, but it will be replaced in future.

We won't list all of the results of the review here, but some highlights include:

- We stabilized the Atomics API.
- We made kotlinx.cinterop Experimental and now require different opt-ins for the package to be used. For more information, see Explicit C-interoperability stability guarantees.
- We marked the Worker class and its related APIs as obsolete.
- We marked the BitSet class as obsolete.
- We marked all public API in the kotlin.native.internal package as private or moved it to other packages.

Explicit C-interoperability stability guarantees

To maintain the high quality of our API, we decided to make kotlinx.cinterop Experimental. Although kotlinx.cinterop has been thoroughly tried and tested, there is still room for improvement before we are satisfied enough to make it Stable. We recommend that you use this API for interoperability but that you try to confine its use to specific areas in your projects. This will make your migration easier once we begin evolving this API to make it Stable.

If you want to use C-like foreign APIs such as pointers, you must opt in with @OptIn(ExperimentalForeignApi), otherwise your code won't compile.

To use the remainder of kotlinx.cinterop, which covers Objective-C/Swift interoperability, you must opt in with @OptIn(BetaInteropApi). If you try to use this API without the opt-in, your code will compile but the compiler will raise warnings that provide a clear explanation of what behavior you can expect.

For more information about these annotations, see our source code for Annotations.kt.

For more information on all of the changes as part of this review, see our YouTrack ticket.

We'd appreciate any feedback you might have! Provide your feedback directly by commenting on the ticket.

Stable @Volatile annotation

If you annotate a var property with @Volatile, then the backing field is marked so that any reads or writes to this field are atomic, and writes are always made visible to other threads.

Prior to 1.8.20, the kotlin.jvm.Volatile annotation was available in the common standard library. However, this annotation was only effective on the JVM. If you used it on other platforms, it was ignored, which leads to errors.

In 1.8.20, we introduced an experimental common annotation, kotlin.concurrent.Volatile, which you could preview in both the JVM and Kotlin/Native.

In 1.9.0, kotlin.concurrent.Volatile is Stable. If you use kotlin.jvm.Volatile in your multiplatform projects, we recommend that you migrate to kotlin.concurrent.Volatile.

New common function to get regex capture group by name

Prior to 1.9.0, every platform had its own extension to get a regular expression capture group by its name from a regular expression match. However there was no common function. It wasn't possible to have a common function prior to Kotlin 1.8.0, because the standard library still supported JVM targets 1.6 and 1.7.

As of Kotlin 1.8.0, the standard library is compiled with JVM target 1.8. So in 1.9.0, there is now a common function groups that you can use to retrieve a group's contents by its name for a regular expression match. This is useful when you want to access the results of regular expression matches belonging to a particular capture group.

Here is an example with a regular expression containing three capture groups: city, state, and areaCode. You can use these group names to access the matched values:

```
fun main() {
    val regex = """\b(?<city>[A-Za-z\s]+),\s(?<state>[A-Z]{2}):\s(?<areaCode>[0-9]{3})\b""".toRegex()
    val input = "Coordinates: Austin, TX: 123"

    val match = regex.find(input)!!
    println(match.groups["city"]?.value)
    // Austin
    println(match.groups["state"]?.value)
    // TX
    println(match.groups["areaCode"]?.value)
    // 123
}
```

New path utility to create parent directories

In 1.9.0 there is a new extension function createParentDirectories() that you can use to create a new file with all the necessary parent directories. When you provide a file path to createParentDirectories() it checks if the parent directories already exist. If they do, it does nothing. However, if they do not, it creates them for you.

createParentDirectories() is particularly useful when you are copying files. For example, you can use it in combination with the copyToRecursively() function:

```
sourcePath.copyToRecursively(
   destinationPath.createParentDirectories(),
   followLinks = false
)
```

New HexFormat class to format and parse hexadecimals

> The new HexFormat class and its related extension functions are Experimental, and to use them, you can opt in with @OptIn(ExperimentalStdlibApi::class) or the compiler argument -opt-in=kotlin.ExperimentalStdlibApi.

In 1.9.0, the HexFormat class and its related extension functions are provided as an Experimental feature that allows you to convert between numerical values and hexadecimal strings. Specifically, you can use the extension functions to convert between hexadecimal strings and ByteArrays or other numeric types (Int, Short, Long).

For example:

```kotlin
println(93.toHexString()) // "0000005d"
```

The HexFormat class includes formatting options that you can configure with the HexFormat{} builder.

If you are working with ByteArrays you have the following options configurable by properties:

Option	Description
upperCase	Whether hexadecimal digits are upper or lower case. By default, lower case is assumed. upperCase = false.
bytes.bytesPerLine	The maximum number of bytes per line.
bytes.bytesPerGroup	The maximum number of bytes per group.
bytes.bytesSeparator	The separator between bytes. Nothing by default.
bytes.bytesPrefix	The string that immediately precedes two-digit hexadecimal representation of each byte. Nothing by default.
bytes.bytesSuffix	The string that immediately succeeds two-digit hexadecimal representation of each byte. Nothing by default.

For example:

```kotlin
val macAddress = "001b638445e6".hexToByteArray()

// Use HexFormat{} builder to separate the hexadecimal string by colons
println(macAddress.toHexString(HexFormat { bytes.byteSeparator = ":" }))
// "00:1b:63:84:45:e6"

// Use HexFormat{} builder to:
// * Make the hexadecimal string uppercase
// * Group the bytes in pairs
// * Separate by periods
val threeGroupFormat = HexFormat { upperCase = true; bytes.bytesPerGroup = 2; bytes.groupSeparator = "." }

println(macAddress.toHexString(threeGroupFormat))
// "001B.6384.45E6"
```

If you are working with numeric types, you have the following options configurable by properties:

Option	Description
number.prefix	The prefix of a hexadecimal string. Nothing by default.
number.suffix	The suffix of a hexadecimal string. Nothing by default.
number.removeLeadingZeros	Whether to remove leading zeros in a hexadecimal string. By default, no leading zeros are removed. number.removeLeadingZeros = false

For example:

```kotlin
// Use HexFormat{} builder to parse a hexadecimal that has prefix: "0x".
println("0x3a".hexToInt(HexFormat { number.prefix = "0x" })) // "58"
```

Documentation updates

The Kotlin documentation has received some notable changes:

- The tour of Kotlin – learn the fundamentals of the Kotlin programming language with chapters including both theory and practice.
- Android source set layout – learn about the new Android source set layout.
- Compatibility guide for Kotlin Multiplatform – learn about the incompatible changes you might encounter while developing projects with Kotlin Multiplatform.
- Kotlin Wasm – learn about Kotlin/Wasm and how you can use it in your Kotlin Multiplatform projects.
- Add dependencies on Kotlin libraries to Kotlin/Wasm project – learn about the supported Kotlin libraries for Kotlin/Wasm.

Install Kotlin 1.9.0

Check the IDE version

IntelliJ IDEA 2022.3.3 and 2023.1.1 automatically suggest updating the Kotlin plugin to version 1.9.0. IntelliJ IDEA 2023.2 will include the built-in Kotlin 1.9.0 plugin.

Android Studio Giraffe (223) and Hedgehog (231) will support Kotlin 1.9.0 in their upcoming releases.

The new command-line compiler is available for download on the GitHub release page.

Configure Gradle settings

To download Kotlin artifacts and dependencies, update your settings.gradle(.kts) file to use the Maven Central repository:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

If the repository is not specified, Gradle uses the sunset JCenter repository, which could lead to issues with Kotlin artifacts.

Compatibility guide for Kotlin 1.9.0

Kotlin 1.9.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of these changes in the Compatibility guide for Kotlin 1.9.0.

What's new in Kotlin 1.8.20

Released: 25 April 2023

The Kotlin 1.8.20 release is out and here are some of its biggest highlights:

- New Kotlin K2 compiler updates
- New experimental Kotlin/Wasm target
- New JVM incremental compilation by default in Gradle
- Update for Kotlin/Native targets
- Preview of Gradle composite builds in Kotlin Multiplatform
- Improved output for Gradle errors in Xcode
- Experimental support for the AutoCloseable interface in the standard library
- Experimental support for Base64 encoding in the standard library

You can also find a short overview of the changes in this video:

Watch video online.

IDE support

The Kotlin plugins that support 1.8.20 are available for:

IDE	Supported versions
IntelliJ IDEA	2022.2.x, 2022.3.x, 2023.1.x
Android Studio	Flamingo (222)

To download Kotlin artifacts and dependencies properly, configure Gradle settings to use the Maven Central repository.

New Kotlin K2 compiler updates

The Kotlin team continues to stabilize the K2 compiler. As mentioned in the Kotlin 1.7.0 announcement, it's still in Alpha. This release introduces further improvements on the road to K2 Beta.

Starting with this 1.8.20 release, the Kotlin K2 compiler:

- Has a preview version of the serialization plugin.
- Provides Alpha support for the JS IR compiler.
- Introduces the future release of the new language version, Kotlin 2.0.

Learn more about the new compiler and its benefits in the following videos:

- What Everyone Must Know About The NEW Kotlin K2 Compiler
- The New Kotlin K2 Compiler: Expert Review

How to enable the Kotlin K2 compiler

To enable and test the Kotlin K2 compiler, use the new language version with the following compiler option:

```
-language-version 2.0
```

You can specify it in your build.gradle(.kts) file:

```
kotlin {
   sourceSets.all {
       languageSettings {
           languageVersion = "2.0"
       }
   }
}
```

The previous -Xuse-k2 compiler option has been deprecated.

> The Alpha version of the new K2 compiler only works with JVM and JS IR projects. It doesn't support Kotlin/Native or any of the multiplatform projects yet.

Leave your feedback on the new K2 compiler

We would appreciate any feedback you may have!

- Provide your feedback directly to K2 developers on Kotlin Slack – get an invite and join the #k2-early-adopters channel.
- Report any problems you faced with the new K2 compiler on our issue tracker.
- Enable the Send usage statistics option to allow JetBrains to collect anonymous data about K2 usage.

Language

As Kotlin continues to evolve, we're introducing preview versions for new language features in 1.8.20:

- A modern and performant replacement of the Enum class values function
- Data objects for symmetry with data classes
- Lifting restrictions on secondary constructors with bodies in inline classes

A modern and performant replacement of the Enum class values function

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Enum classes have a synthetic values() function, which returns an array of defined enum constants. However, using an array can lead to hidden performance issues in Kotlin and Java. In addition, most of the APIs use collections, which require eventual conversion. To fix these problems, we've introduced the entries property for Enum classes, which should be used instead of the values() function. When called, the entries property returns a pre-allocated immutable list of defined enum constants.

> The values() function is still supported, but we recommend that you use the entries property instead.

```
enum class Color(val colorName: String, val rgb: String) {
   RED("Red", "#FF0000"),
   ORANGE("Orange", "#FF7F00"),
   YELLOW("Yellow", "#FFFF00")
}


@OptIn(ExperimentalStdlibApi::class)
fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb == rgb }
```

How to enable the entries property

To try this feature out, opt in with @OptIn(ExperimentalStdlibApi) and enable the -language-version 1.9 compiler option. In a Gradle project, you can do so by adding the following to your build.gradle(.kts):

Kotlin

```kotlin
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```groovy
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
}
```

> Starting with IntelliJ IDEA 2023.1, if you have opted in to this feature, the appropriate IDE inspection will notify you about converting from values() to entries and offer a quick-fix.

For more information on the proposal, see the KEEP note.

Preview of data objects for symmetry with data classes

Data objects allow you to declare objects with singleton semantics and a clean toString() representation. In this snippet, you can see how adding the data keyword to an object declaration improves the readability of its toString() output:

```kotlin
package org.example
object MyObject
data object MyDataObject

fun main() {
    println(MyObject) // org.example.MyObject@1f32e575
    println(MyDataObject) // MyDataObject
}
```

Especially for sealed hierarchies (like a sealed class or sealed interface hierarchy), data objects are an excellent fit because they can be used conveniently alongside data class declarations. In this snippet, declaring EndOfFile as a data object instead of a plain object means that it will get a pretty toString without the need to override it manually. This maintains symmetry with the accompanying data class definitions.

```kotlin
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // Number(number=7)
    println(EndOfFile) // EndOfFile
}
```

Semantics of data objects

Since their first preview version in Kotlin 1.7.20, the semantics of data objects have been refined. The compiler now automatically generates a number of convenience functions for them:

toString

The toString() function of a data object returns the simple name of the object:

```
data object MyDataObject {
    val x: Int = 3
}

fun main() {
    println(MyDataObject) // MyDataObject
}
```

equals and hashCode

The equals() function for a data object ensures that all objects that have the type of your data object are considered equal. In most cases, you will only have a single instance of your data object at runtime (after all, a data object declares a singleton). However, in the edge case where another object of the same type is generated at runtime (for example, via platform reflection through java.lang.reflect, or by using a JVM serialization library that uses this API under the hood), this ensures that the objects are treated as equal.

Make sure to only compare data objects structurally (using the == operator) and never by reference (the === operator). This helps avoid pitfalls when more than one instance of a data object exists at runtime. The following snippet illustrates this specific edge case:

```
import java.lang.reflect.Constructor

data object MySingleton

fun main() {
    val evilTwin = createInstanceViaReflection()

    println(MySingleton) // MySingleton
    println(evilTwin) // MySingleton

    // Even when a library forcefully creates a second instance of MySingleton, its `equals` method returns true:
    println(MySingleton == evilTwin) // true

    // Do not compare data objects via ===.
    println(MySingleton === evilTwin) // false
}

fun createInstanceViaReflection(): MySingleton {
    // Kotlin reflection does not permit the instantiation of data objects.
    // This creates a new MySingleton instance "by force" (i.e., Java platform reflection)
    // Don't do this yourself!
    return (MySingleton.javaClass.declaredConstructors[0].apply { isAccessible = true } as Constructor<MySingleton>).newInstance()
}
```

The behavior of the generated hashCode() function is consistent with that of the equals() function, so that all runtime instances of a data object have the same hash code.

No copy and componentN functions for data objects

While data object and data class declarations are often used together and have some similarities, there are some functions that are not generated for a data object:

Because a data object declaration is intended to be used as a singleton object, no copy() function is generated. The singleton pattern restricts the instantiation of a class to a single instance, and allowing copies of the instance to be created would violate that restriction.

Also, unlike a data class, a data object does not have any data properties. Since attempting to destructure such an object would not make sense, no componentN() functions are generated.

We would appreciate your feedback on this feature in YouTrack.

How to enable the data objects preview

To try this feature out, enable the -language-version 1.9 compiler option. In a Gradle project, you can do so by adding the following to your build.gradle(.kts):

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
```

```
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
}
```

Preview of lifting restriction on secondary constructors with bodies in inline classes

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.8.20 lifts restrictions on the use of secondary constructors with bodies in inline classes.

Inline classes used to allow only a public primary constructor without init blocks or secondary constructors to have clear initialization semantics. As a result, it was impossible to encapsulate underlying values or create an inline class that would represent some constrained values.

These issues were fixed when Kotlin 1.4.30 lifted restrictions on init blocks. Now we're taking it a step further and allowing secondary constructors with bodies in preview mode:

```
@JvmInline
value class Person(private val fullName: String) {
// Allowed since Kotlin 1.4.30:
    init {
        check(fullName.isNotBlank()) {
            "Full name shouldn't be empty"
        }
    }
// Preview available since Kotlin 1.8.20:
    constructor(name: String, lastName: String) : this("$name $lastName") {
        check(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }
}
```

How to enable secondary constructors with bodies

To try this feature out, enable the -language-version 1.9 compiler option. In a Gradle project, you can do so by adding the following to your build.gradle(.kts):

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
```

```
}
```

We encourage you to try this feature out and submit all reports in YouTrack to help us make it the default in Kotlin 1.9.0.

Learn more about the development of Kotlin inline classes in this KEEP.

New Kotlin/Wasm target

Kotlin/Wasm (Kotlin WebAssembly) goes Experimental in this preview release. The Kotlin team finds WebAssembly to be a promising technology and wants to find better ways for you to use it and get all of the benefits of Kotlin.

WebAssembly binary format is independent of the platform because it runs using its own virtual machine. Almost all modern browsers already support WebAssembly 1.0. To set up the environment to run WebAssembly, you only need to enable an experimental garbage collection mode that Kotlin/Wasm targets. You can find detailed instructions here: How to enable Kotlin/Wasm.

We want to highlight the following advantages of the new Kotlin/Wasm target:

- Faster compilation speed compared to the wasm32 Kotlin/Native target, since Kotlin/Wasm doesn't have to use LLVM.
- Easier interoperability with JS and integration with browsers compared to the wasm32 target, thanks to the Wasm garbage collection.
- Potentially faster application startup compared to Kotlin/JS and JavaScript because Wasm has a compact and easy-to-parse bytecode.
- Improved application runtime performance compared to Kotlin/JS and JavaScript because Wasm is a statically typed language.

Starting with the 1.8.20-RC2 preview release, you can use Kotlin/Wasm in your experimental projects. We provide the Kotlin standard library (stdlib) and test library (kotlin.test) for Kotlin/Wasm out of the box. IDE support will be added in future releases.

Learn more about Kotlin/Wasm in this YouTube video.

How to enable Kotlin/Wasm

To enable and test Kotlin/Wasm, update your build.gradle.kts file:

```
plugins {
    kotlin("multiplatform") version "1.8.20-RC2"
}

kotlin {
    wasm {
        binaries.executable()
        browser {
        }
    }
    sourceSets {
        val commonMain by getting
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
        val wasmMain by getting
        val wasmTest by getting
    }
}
```

> Check out the GitHub repository with Kotlin/Wasm examples.

To run a Kotlin/Wasm project, you need to update the settings of the target environment:

Chrome

- For version 109:

  Run the application with the --js-flags=--experimental-wasm-gc command line argument.

- For version 110 or later:

1. Go to chrome://flags/#enable-webassembly-garbage-collection in your browser.
2. Enable WebAssembly Garbage Collection.
3. Relaunch your browser.

Firefox

For version 109 or later:

1. Go to about:config in your browser.
2. Enable javascript.options.wasm_function_references and javascript.options.wasm_gc options.
3. Relaunch your browser.

Edge

For version 109 or later:

Run the application with the --js-flags=--experimental-wasm-gc command line argument.

Leave your feedback on Kotlin/Wasm

We would appreciate any feedback you may have!

- Provide your feedback directly to developers in Kotlin Slack – get an invite and join the #webassembly channel.
- Report any problems you faced with Kotlin/Wasm on this YouTrack issue.

Kotlin/JVM

Kotlin 1.8.20 introduces a preview of Java synthetic property references and support for the JVM IR backend in the kapt stub generating task by default.

Preview of Java synthetic property references

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.8.20 introduces the ability to create references to Java synthetic properties, for example, for such Java code:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Kotlin has always allowed you to write person.age, where age is a synthetic property. Now, you can also create references to Person::age and person::age. All the same works for name, as well.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12), Person("Peter", 11))
    Persons
        // Call a reference to Java synthetic property:
        .sortedBy(Person::age)
        // Call Java getter via the Kotlin property syntax:
        .forEach { person -> println(person.name) }
}
```

How to enable Java synthetic property references

To try this feature out, enable the -language-version 1.9 compiler option. In a Gradle project, you can do so by adding the following to your build.gradle(.kts):

Kotlin

```kotlin
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```groovy
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
}
```

Support for the JVM IR backend in kapt stub generating task by default

In Kotlin 1.7.20, we introduced support for the JVM IR backend in the kapt stub generating task. Starting with this release, this support works by default. You no longer need to specify kapt.use.jvm.ir=true in your gradle.properties to enable it. We would appreciate your feedback on this feature in YouTrack.

Kotlin/Native

Kotlin 1.8.20 includes changes to supported Kotlin/Native targets, interoperability with Objective-C, and improvements to the CocoaPods Gradle plugin, among other updates:

- Update for Kotlin/Native targets
- Deprecation of the legacy memory manager
- Support for Objective-C headers with @import directives
- Support for link-only mode in the Cocoapods Gradle plugin
- Import Objective-C extensions as class members in UIKit
- Reimplementation of compiler cache management in the compiler
- Deprecation of useLibraries() in Cocoapods Gradle plugin

Update for Kotlin/Native targets

The Kotlin team decided to revisit the list of targets supported by Kotlin/Native, split them into tiers, and deprecate some of them starting with Kotlin 1.8.20. See the Kotlin/Native target support section for the full list of supported and deprecated targets.

The following targets have been deprecated with Kotlin 1.8.20 and will be removed in 1.9.20:

- iosArm32
- watchosX86
- wasm32
- mingwX86
- linuxArm32Hfp

- linuxMips32
- linuxMipsel32

As for the remaining targets, there are now three tiers of support depending on how well a target is supported and tested in the Kotlin/Native compiler. A target can be moved to a different tier. For example, we'll do our best to provide full support for iosArm64 in the future, as it is important for Kotlin Multiplatform Mobile.

If you're a library author, these target tiers can help you decide which targets to test on CI tools and which ones to skip. The Kotlin team will use the same approach when developing official Kotlin libraries, like kotlinx.coroutines.

Check out our blog post to learn more about the reasons for these changes.

Deprecation of the legacy memory manager

Starting with 1.8.20, the legacy memory manager is deprecated and will be removed in 1.9.20. The new memory manager was enabled by default in 1.7.20 and has been receiving further stability updates and performance improvements.

If you're still using the legacy memory manager, remove the kotlin.native.binary.memoryModel=strict option from your gradle.properties and follow our Migration guide to make the necessary changes.

The new memory manager doesn't support the wasm32 target. This target is also deprecated starting with this release and will be removed in 1.9.20.

Support for Objective-C headers with @import directives

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin/Native can now import Objective-C headers with @import directives. This feature is useful for consuming Swift libraries that have auto-generated Objective-C headers or classes of CocoaPods dependencies written in Swift.

Previously, the cinterop tool failed to analyze headers that depended on Objective-C modules via the @import directive. The reason was that it lacked support for the -fmodules option.

Starting with Kotlin 1.8.20, you can use Objective-C headers with @import. To do so, pass the -fmodules option to the compiler in the definition file as compilerOpts. If you use CocoaPods integration, specify the cinterop option in the configuration block of the pod() function like this:

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("PodName") {
            extraOpts = listOf("-compiler-option", "-fmodules")
        }
    }
}
```

This was a highly awaited feature, and we welcome your feedback about it in YouTrack to help us make it the default in future releases.

Support for the link-only mode in Cocoapods Gradle plugin

With Kotlin 1.8.20, you can use Pod dependencies with dynamic frameworks only for linking, without generating C-interop bindings. This may come in handy when C-interop bindings are already generated.

Consider a project with 2 modules, a library and an app. The library depends on a Pod but doesn't produce a framework, only a .klib. The app depends on the library and produces a dynamic framework. In this case, you need to link this framework with the Pods that the library depends on, but you don't need C-interop bindings because they are already generated for the library.

To enable the feature, use the linkOnly option or a builder property when adding a dependency on a Pod:

```
cocoapods {
```

```
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        pod("AFNetworking", linkOnly = true) {
            version = "~> 4.0.0"
        }
    }
```

If you use this option with static frameworks, it will remove the Pod dependency entirely because Pods are not used for static framework linking.

Import Objective-C extensions as class members in UIKit

Since Xcode 14.1, some methods from Objective-C classes have been moved to category members. That led to the generation of a different Kotlin API, and these methods were imported as Kotlin extensions instead of methods.

You may have experienced issues resulting from this when overriding methods using UIKit. For example, it became impossible to override drawRect() or layoutSubviews() methods when subclassing a UIVIew in Kotlin.

Since 1.8.20, category members that are declared in the same headers as NSView and UIView classes are imported as members of these classes. This means that the methods subclassing from NSView and UIView can be easily overridden, like any other method.

If everything goes well, we're planning to enable this behavior by default for all of the Objective-C classes.

Reimplementation of compiler cache management in the compiler

To speed up the evolution of compiler caches, we've moved compiler cache management from the Kotlin Gradle plugin to the Kotlin/Native compiler. This unblocks work on several important improvements, including those to do with compilation times and compiler cache flexibility.

If you encounter some problem and need to return to the old behavior, use the kotlin.native.cacheOrchestration=gradle Gradle property.

We would appreciate your feedback on this in YouTrack.

Deprecation of useLibraries() in Cocoapods Gradle plugin

Kotlin 1.8.20 starts the deprecation cycle of the useLibraries() function used in the CocoaPods integration for static libraries.

We introduced the useLibraries() function to allow dependencies on Pods containing static libraries. With time, this case has become very rare. Most of the Pods are distributed by sources, and Objective-C frameworks or XCFrameworks are a common choice for binary distribution.

Since this function is unpopular and it creates issues that complicate the development of the Kotlin CocoaPods Gradle plugin, we've decided to deprecate it.

For more information on frameworks and XCFrameworks, see Build final native binaries.

Kotlin Multiplatform

Kotlin 1.8.20 strives to improve the developer experience with the following updates to Kotlin Multiplatform:

- New approach for setting up source set hierarchy
- Preview of Gradle composite builds support in Kotlin Multiplatform
- Improved output for Gradle errors in Xcode

New approach to source set hierarchy

The new approach to source set hierarchy is Experimental. It may be changed in future Kotlin releases without prior notice. Opt-in is required (see the details below). We would appreciate your feedback in YouTrack.

Kotlin 1.8.20 offers a new way of setting up source set hierarchy in your multiplatform projects – the default target hierarchy. The new approach is intended to replace target shortcuts like ios, which have their design flaws.

The idea behind the default target hierarchy is simple: You explicitly declare all the targets to which your project compiles, and the Kotlin Gradle plugin

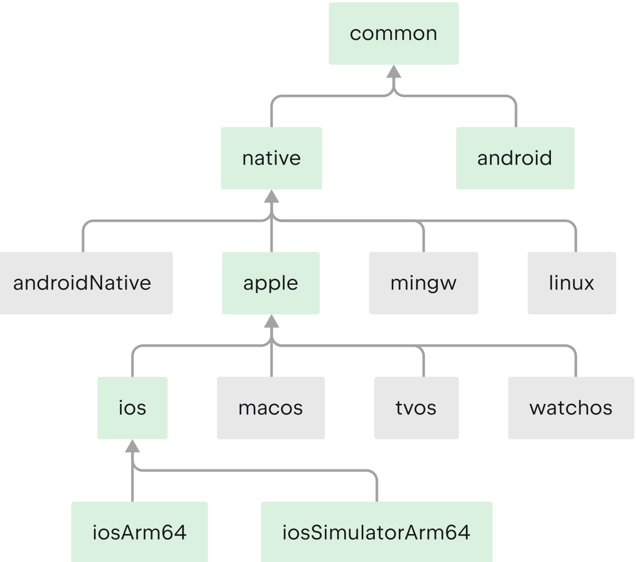
automatically creates shared source sets based on the specified targets.

Set up your project

Consider this example of a simple multiplatform mobile app:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
kotlin {
    // Enable the default target hierarchy:
    targetHierarchy.default()

    android()
    iosArm64()
    iosSimulatorArm64()
}
```

You can think of the default target hierarchy as a template for all possible targets and their shared source sets. When you declare the final targets android, iosArm64, and iosSimulatorArm64 in your code, the Kotlin Gradle plugin finds suitable shared source sets from the template and creates them for you. The resulting hierarchy looks like this:



An example of using the default target hierarchy

Green source sets are actually created and present in the project, while gray ones from the default template are ignored. As you can see, the Kotlin Gradle plugin hasn't created the watchos source set, for example, because there are no watchOS targets in the project.

If you add a watchOS target, such as watchosArm64, the watchos source set is created, and the code from the apple, native, and common source sets is compiled to watchosArm64, as well.

You can find the complete scheme for the default target hierarchy in the documentation.

> In this example, the apple and native source sets compile only to the iosArm64 and iosSimulatorArm64 targets. Therefore, despite their names, they have access to the full iOS API. This might be counter-intuitive for source sets like native, as you may expect that only APIs available on all native targets are accessible in this source set. This behavior may change in the future.

Why replace shortcuts

Creating source sets hierarchies can be verbose, error-prone, and unfriendly for beginners. Our previous solution was to introduce shortcuts like ios that create a part of the hierarchy for you. However, working with shortcuts proved they have a big design flaw: they're difficult to change.

Take the ios shortcut, for example. It creates only the iosArm64 and iosX64 targets, which can be confusing and may lead to issues when working on an M1-based host that requires the iosSimulatorArm64 target as well. However, adding the iosSimulatorArm64 target can be a very disruptive change for user projects:

- All dependencies used in the iosMain source set have to support the iosSimulatorArm64 target; otherwise, the dependency resolution fails.
- Some native APIs used in iosMain may disappear when adding a new target (though this is unlikely in the case of iosSimulatorArm64).
- In some cases, such as when writing a small pet project on your Intel-based MacBook, you might not even need this change.

It became clear that shortcuts didn't solve the problem of configuring hierarchies, which is why we stopped adding new shortcuts at some point.

The default target hierarchy may look similar to shortcuts at first glance, but they have a crucial distinction: users have to explicitly specify the set of targets. This set defines how your project is compiled and published and how it participates in dependency resolution. Since this set is fixed, changes to the default configuration from the Kotlin Gradle plugin should cause significantly less distress in the ecosystem, and it will be much easier to provide tooling-assisted migration.

How to enable the default hierarchy

This new feature is Experimental. For Kotlin Gradle build scripts, you need to opt in with @OptIn(ExperimentalKotlinGradlePluginApi::class).

For more information, see Hierarchical project structure.

Leave feedback

This is a significant change to multiplatform projects. We would appreciate your feedback to help make it even better.

Preview of Gradle composite builds support in Kotlin Multiplatform

> This feature has been supported in Gradle builds since Kotlin Gradle Plugin 1.8.20. For IDE support, use IntelliJ IDEA 2023.1 Beta 2 (231.8109.2) or later and the Kotlin Gradle plugin 1.8.20 with any Kotlin IDE plugin.

Starting with 1.8.20-RC2, Kotlin Multiplatform supports Gradle composite builds. Composite builds allow you to include builds of separate projects or parts of the same project into a single build.

Due to some technical challenges, using Gradle composite builds with Kotlin Multiplatform was only partially supported. Kotlin 1.8.20-RC2 contains a preview of the improved support that should work with a larger variety of projects. To try it out, add the following option to your gradle.properties:

```
kotlin.mpp.import.enableKgpDependencyResolution=true
```

This option enables a preview of the new import mode. Besides the support for composite builds, it provides a smoother import experience in multiplatform projects, as we've included major bug fixes and improvements to make the import more stable.

Known issues

It's still a preview version that needs further stabilization, and you might encounter some issues with import along the way. Here are some known issues we're planning to fix before the final release of Kotlin 1.8.20:

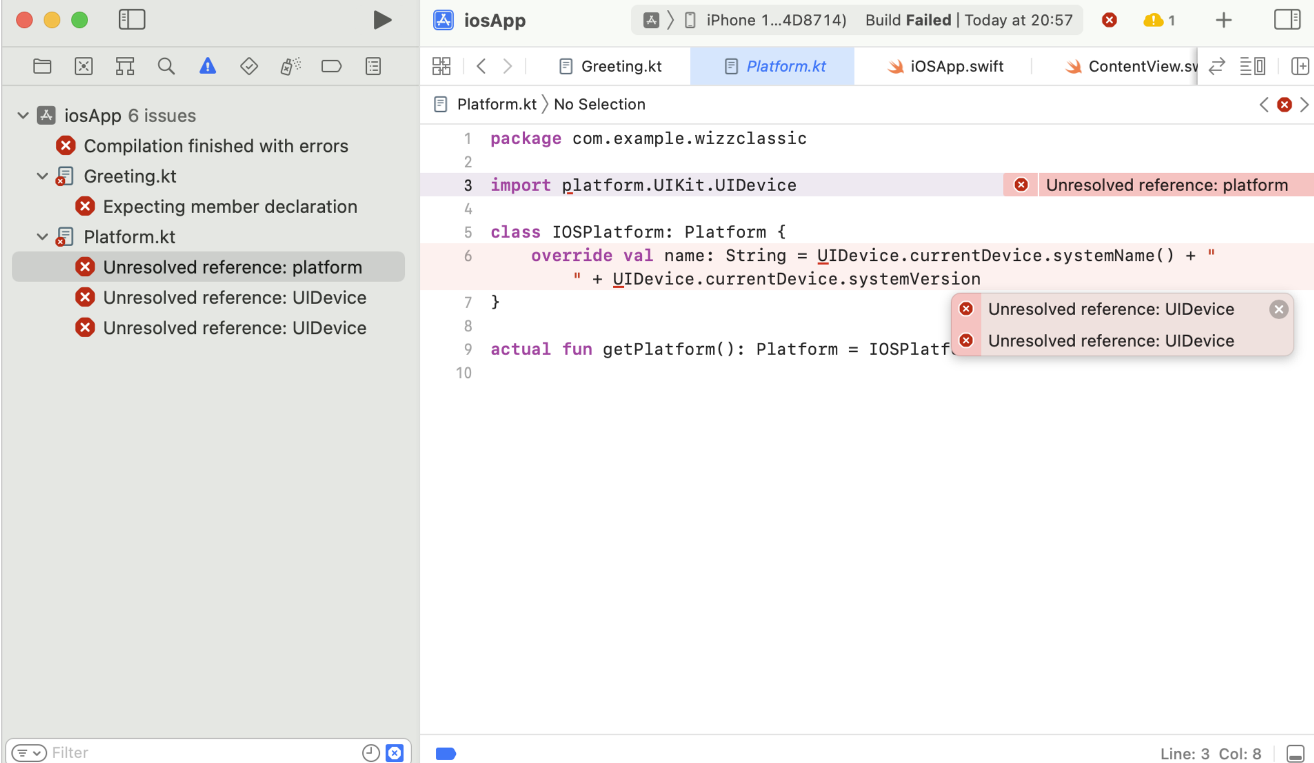
- There's no Kotlin 1.8.20 plugin available for IntelliJ IDEA 2023.1 EAP yet. Despite that, you can still set the Kotlin Gradle plugin version to 1.8.20-RC2 and try out composite builds in this IDE.
- If your projects include builds with a specified rootProject.name, composite builds may fail to resolve the Kotlin metadata. For the workaround and details, see this Youtrack issue.

We encourage you to try it out and submit all reports on YouTrack to help us make it the default in Kotlin 1.9.0.

Improved output for Gradle errors in Xcode

If you had issues building your multiplatform projects in Xcode, you might have encountered a "Command PhaseScriptExecution failed with a nonzero exit code" error. This message signals that the Gradle invocation has failed, but it's not very helpful when trying to detect the problem.

Starting with Kotlin 1.8.20-RC2, Xcode can parse the output from the Kotlin/Native compiler. Furthermore, in case the Gradle build fails, you'll see an additional error message from the root cause exception in Xcode. In most cases, it'll help to identify the root problem.



Improved output for Gradle errors in Xcode

The new behavior is enabled by default for the standard Gradle tasks for Xcode integration, like embedAndSignAppleFrameworkForXcode that can connect the iOS framework from your multiplatform project to the iOS application in Xcode. It can also be enabled (or disabled) with the kotlin.native.useXcodeMessageStyle Gradle property.

Kotlin/JavaScript

Kotlin 1.8.20 changes the ways TypeScript definitions can be generated. It also includes a change designed to improve your debugging experience:

- Removal of Dukat integration from the Gradle plugin
- Kotlin variable and function names in source maps
- Opt in for generation of TypeScript definition files

Removal of Dukat integration from Gradle plugin

In Kotlin 1.8.20, we've removed our Experimental Dukat integration from the Kotlin/JavaScript Gradle plugin. The Dukat integration supported the automatic conversion of TypeScript declaration files (.d.ts) into Kotlin external declarations.

You can still convert TypeScript declaration files (.d.ts) into Kotlin external declarations by using our Dukat tool instead.

> The Dukat tool is Experimental. It may be dropped or changed at any time.

Kotlin variable and function names in source maps

To help with debugging, we've introduced the ability to add the names that you declared in Kotlin code for variables and functions into your source maps. Prior to 1.8.20, these weren't available in source maps, so in the debugger, you always saw the variable and function names of the generated JavaScript.

You can configure what is added by using sourceMapNamesPolicy in your Gradle file build.gradle.kts, or the -source-map-names-policy compiler option. The table below lists the possible settings:

Setting	Description	Example output
simple-names	Variable names and simple function names are added. (Default)	main
fully-qualified-names	Variable names and fully qualified function names are added.	com.example.kjs.playground.main
no	No variable or function names are added.	N/A

See below for an example configuration in a build.gradle.kts file:

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.Kotlin2JsCompile>().configureEach {

compilercompileOptions.sourceMapNamesPolicy.set(org.jetbrains.kotlin.gradle.dsl.JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_FQ_NAMES
// or SOURCE_MAP_NAMES_POLICY_NO, or SOURCE_MAP_NAMES_POLICY_SIMPLE_NAMES
}
```

Debugging tools like those provided in Chromium-based browsers can pick up the original Kotlin names from your source map to improve the readability of your stack trace. Happy debugging!

> The addition of variable and function names in source maps is Experimental. It may be dropped or changed at any time.

Opt in for generation of TypeScript definition files

Previously, if you had a project that produced executable files (binaries.executable()), the Kotlin/JS IR compiler collected any top-level declarations marked with @JsExport and automatically generated TypeScript definitions in a .d.ts file.

As this isn't useful for every project, we've changed the behavior in Kotlin 1.8.20. If you want to generate TypeScript definitions, you have to explicitly configure this in your Gradle build file. Add generateTypeScriptDefinitions() to your build.gradle.kts.file in the js section. For example:

```
kotlin {
   js {
       binaries.executable()
       browser {
       }
       generateTypeScriptDefinitions()
   }
}
```

> The generation of TypeScript definitions (d.ts) is Experimental. It may be dropped or changed at any time.

Gradle

Kotlin 1.8.20 is fully compatible with Gradle 6.8 through 7.6 except for some special cases in the Multiplatform plugin. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings the following changes:

- New alignment of Gradle plugins' versions
- New JVM incremental compilation by default in Gradle
- Precise backup of compilation tasks' outputs
- Lazy Kotlin/JVM task creation for all Gradle versions
- Non-default location of compile tasks' destinationDirectory

- Ability to opt-out from reporting compiler arguments to an HTTP statistics service

New Gradle plugins versions alignment

Gradle provides a way to ensure dependencies that must work together are always aligned in their versions. Kotlin 1.8.20 adopted this approach, too. It works by default so that you don't need to change or update your configuration to enable it. In addition, you no longer need to resort to this workaround for resolving Kotlin Gradle plugins' transitive dependencies.

We would appreciate your feedback on this feature in YouTrack.

New JVM incremental compilation by default in Gradle

The new approach to incremental compilation, which has been available since Kotlin 1.7.0, now works by default. You no longer need to specify kotlin.incremental.useClasspathSnapshot=true in your gradle.properties to enable it.

We would appreciate your feedback on this. You can file an issue in YouTrack.

Precise backup of compilation tasks' outputs

> Precise backup of compilation tasks' outputs is Experimental. To use it, add kotlin.compiler.preciseCompilationResultsBackup=true to gradle.properties. We would appreciate your feedback on it in YouTrack.

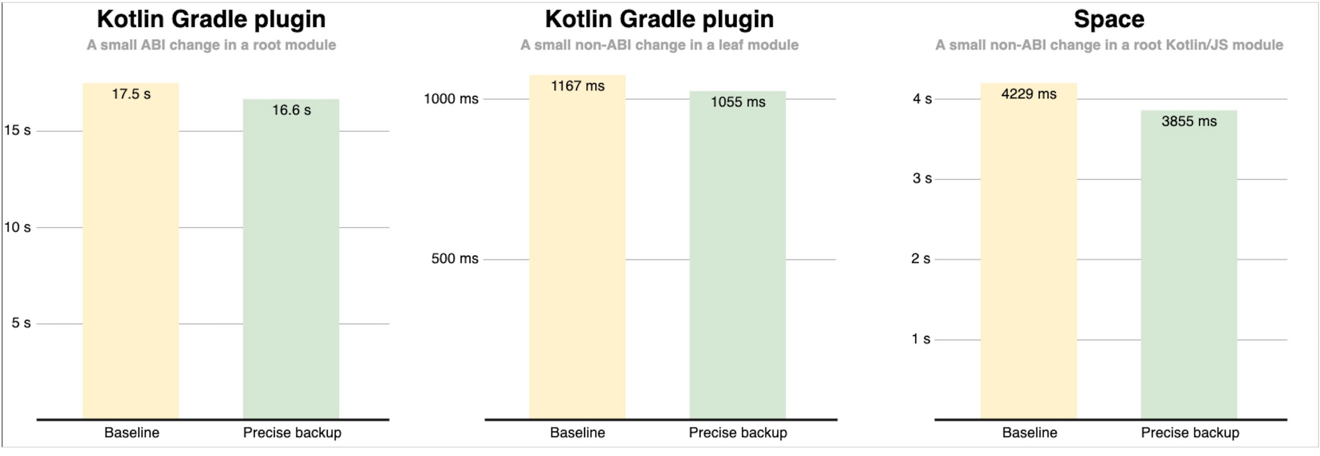
Starting with Kotlin 1.8.20, you can enable precise backup, whereby only those classes that Kotlin recompiles in the incremental compilation will be backed up. Both full and precise backups help to run builds incrementally again after compilation errors. Precise backup also saves build time compared to full backup. Full backup may take noticeable build time in large projects or if many tasks are making backups, especially if a project is located on a slow HDD.

This optimization is Experimental. You can enable it by adding the kotlin.compiler.preciseCompilationResultsBackup Gradle property to the gradle.properties file:

```
kotlin.compiler.preciseCompilationResultsBackup=true
```

Example of precise backup usage in JetBrains

In the following charts, you can see examples of using precise backup compared to full backup:



Comparison of full and precise backups

The first and second charts show how precise backup in the Kotlin project affects building the Kotlin Gradle plugin:

1. After making a small ABI change – adding a new public method – to a module that lots of modules depend on.
2. After making a small non-ABI change – adding a private function – to a module that no other modules depend on.

The third chart shows how precise backup in the Space project affects building a web frontend after a small non-ABI change – adding a private function – to a Kotlin/JS module that lots of modules depend on.

These measurements were performed on a computer with the Apple M1 Max CPU; different computers will yield slightly different results. The factors affecting performance include but are not limited to:

- How warm the Kotlin daemon and the Gradle daemon are.
- How fast or slow the disk is.
- The CPU model and how busy it is.
- Which modules are affected by the changes and how big these modules are.
- Whether the changes are ABI or non-ABI.

Evaluating optimizations with build reports

To estimate the impact of the optimization on your computer for your project and your scenarios, you can use Kotlin build reports. Enable reports in the text file format by adding the following property to your gradle.properties file:

```
kotlin.build.report.output=file
```

Here is an example of a relevant part of the report before enabling precise backup:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.59 s
<...>
Time metrics:
 Total Gradle task time: 0.59 s
 Task action before worker execution: 0.24 s
  Backup output: 0.22 s // Pay attention to this number
<...>
```

And here is an example of a relevant part of the report after enabling precise backup:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.46 s
<...>
Time metrics:
 Total Gradle task time: 0.46 s
 Task action before worker execution: 0.07 s
  Backup output: 0.05 s // The time has reduced
 Run compilation in Gradle worker: 0.32 s
  Clear jar cache: 0.00 s
  Precise backup output: 0.00 s // Related to precise backup
  Cleaning up the backup stash: 0.00 s // Related to precise backup
<...>
```

Lazy Kotlin/JVM tasks creation for all Gradle versions

For projects with the "org.jetbrains.kotlin.gradle.jvm" plugin on Gradle 7.3+, the Kotlin Gradle plugin no longer creates and configures the task "compileKotlin" eagerly. On lower Gradle versions, it simply registers all the tasks and doesn't configure them on a dry run. The same behavior is now in place when using Gradle 7.3+.

Non-default location of compile tasks' destinationDirectory

Update your build script with some additional code if you do one of the following:

- Override the Kotlin/JVM KotlinJvmCompile/KotlinCompile task's destinationDirectory location.
- Use a deprecated Kotlin/JS/Non-IR variant and override the Kotlin2JsCompile task's destinationDirectory.

You need to explicitly add sourceSets.main.kotlin.classesDirectories to sourceSets.main.outputs in your JAR file:

tasks.jar(type: Jar) { from sourceSets.main.outputs from sourceSets.main.kotlin.classesDirectories }

Ability to opt-out from reporting compiler arguments to an HTTP statistics service

You can now control whether the Kotlin Gradle plugin should include compiler arguments in HTTP build reports. Sometimes, you might not need the plugin to report these arguments. If a project contains many modules, its compiler arguments in the report can be very heavy and not that helpful. There is now a way to disable it and thus save memory. In your gradle.properties or local.properties, use the kotlin.build.report.include_compiler_arguments=(true|false) property.

We would appreciate your feedback on this feature on YouTrack.

Standard library

Kotlin 1.8.20 adds a variety of new features, including some that are particularly useful for Kotlin/Native development:

- Support for the AutoCloseable interface
- Support for Base64 encoding and decoding
- Support for @Volatile in Kotlin/Native
- Bug fix for stack overflow when using regex in Kotlin/Native

Support for the AutoCloseable interface

> The new AutoCloseable interface is Experimental, and to use it you need to opt in with @OptIn(ExperimentalStdlibApi::class) or the compiler argument -opt-in=kotlin.ExperimentalStdlibApi.

The AutoCloseable interface has been added to the common standard library so that you can use one common interface for all libraries to close resources. In Kotlin/JVM, the AutoCloseable interface is an alias for java.lang.AutoClosable.

In addition, the extension function use() is now included, which executes a given block function on the selected resource and then closes it down correctly, whether an exception is thrown or not.

There is no public class in the common standard library that implements the AutoCloseable interface. In the example below, we define the XMLWriter interface and assume that there is a resource that implements it. For example, this resource could be a class that opens a file, writes XML content, and then closes it.

```
interface XMLWriter : AutoCloseable {
    fun document(encoding: String, version: String, content: XMLWriter.() -> Unit)
    fun element(name: String, content: XMLWriter.() -> Unit)
    fun attribute(name: String, value: String)
    fun text(value: String)
}

fun writeBooksTo(writer: XMLWriter) {
    writer.use { xml ->
        xml.document(encoding = "UTF-8", version = "1.0") {
            element("bookstore") {
                element("book") {
                    attribute("category", "fiction")
                    element("title") { text("Harry Potter and the Prisoner of Azkaban") }
                    element("author") { text("J. K. Rowling") }
                    element("year") { text("1999") }
                    element("price") { text("29.99") }
                }
                element("book") {
                    attribute("category", "programming")
                    element("title") { text("Kotlin in Action") }
                    element("author") { text("Dmitry Jemerov") }
                    element("author") { text("Svetlana Isakova") }
                    element("year") { text("2017") }
                    element("price") { text("25.19") }
                }
            }
        }
    }
}
```

Support for Base64 encoding

> The new encoding and decoding functionality is Experimental, and to use it, you need to opt in with @OptIn(ExperimentalEncodingApi::class) or the compiler argument -opt-in=kotlin.io.encoding.ExperimentalEncodingApi.

We've added support for Base64 encoding and decoding. We provide 3 class instances, each using different encoding schemes and displaying different behaviors. Use the Base64.Default instance for the standard Base64 encoding scheme.

Use the Base64.UrlSafe instance for the "URL and Filename safe" encoding scheme.

Use the Base64.Mime instance for the MIME encoding scheme. When you use the Base64.Mime instance, all encoding functions insert a line separator every 76 characters. In the case of decoding, any illegal characters are skipped and don't throw an exception.

> The Base64.Default instance is the companion object of the Base64 class. As a result, you can call its functions via Base64.encode() and Base64.decode() instead of Base64.Default.encode() and Base64.Default.decode().

```
val foBytes = "fo".map { it.code.toByte() }.toByteArray()
Base64.Default.encode(foBytes) // "Zm8="
// Alternatively:
// Base64.encode(foBytes)

val foobarBytes = "foobar".map { it.code.toByte() }.toByteArray()
Base64.UrlSafe.encode(foobarBytes) // "Zm9vYmFy"

Base64.Default.decode("Zm8=") // foBytes
// Alternatively:
// Base64.decode("Zm8=")

Base64.UrlSafe.decode("Zm9vYmFy") // foobarBytes
```

You can use additional functions to encode or decode bytes into an existing buffer, as well as to append the encoding result to a provided Appendable type object.

In Kotlin/JVM, we've also added the extension functions encodingWith() and decodingWith() to enable you to perform Base64 encoding and decoding with input and output streams.

Support for @Volatile in Kotlin/Native

> @Volatile in Kotlin/Native is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

If you annotate a var property with @Volatile, then the backing field is marked so that any reads or writes to this field are atomic, and writes are always made visible to other threads.

Prior to 1.8.20, the kotlin.jvm.Volatile annotation was only available in the common standard library. However, this annotation is only effective in the JVM. If you use it in Kotlin/Native, it is ignored, which can lead to errors.

In 1.8.20, we've introduced a common annotation, kotlin.concurrent.Volatile, that you can use in both the JVM and Kotlin/Native.

How to enable

To try this feature out, opt in with @OptIn(ExperimentalStdlibApi) and enable the -language-version 1.9 compiler option. In a Gradle project, you can do so by adding the following to your build.gradle(.kts):

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```kotlin
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
}
```

Bug fix for stack overflow when using regex in Kotlin/Native

In previous versions of Kotlin, a crash could occur if your regex input contained a large number of characters, even when the regex pattern was very simple. In 1.8.20, this issue has been resolved. For more information, see KT-46211.

Serialization updates

Kotlin 1.8.20 comes with Alpha support for the Kotlin K2 compiler and prohibits serializer customization via companion object.

Prototype serialization compiler plugin for Kotlin K2 compiler

> Support for the serialization compiler plugin for K2 is in Alpha. To use it, enable the Kotlin K2 compiler.

Starting with 1.8.20, the serialization compiler plugin works with the Kotlin K2 compiler. Give it a try and share your feedback with us!

Prohibit implicit serializer customization via companion object

Currently, it is possible to declare a class as serializable with the @Serializable annotation and, at the same time, declare a custom serializer with the @Serializer annotation on its companion object.

For example:

```kotlin
import kotlinx.serialization.*

@Serializable
class Foo(val a: Int) {
   @Serializer(Foo::class)
   companion object {
       // Custom implementation of KSerializer<Foo>
   }
}
```

In this case, it's not clear from the @Serializable annotation which serializer is used. In actual fact, class Foo has a custom serializer.

To prevent this kind of confusion, in Kotlin 1.8.20 we've introduced a compiler warning for when this scenario is detected. The warning includes a possible migration path to resolve this issue.

If you use such constructs in your code, we recommend updating them to the below:

```kotlin
import kotlinx.serialization.*

@Serializable(Foo.Companion::class)
class Foo(val a: Int) {
   // Doesn't matter if you use @Serializer(Foo::class) or not
   companion object: KSerializer<Foo> {
       // Custom implementation of KSerializer<Foo>
   }
}
```

With this approach, it is clear that the Foo class uses the custom serializer declared in the companion object. For more information, see our YouTrack ticket.

> In Kotlin 2.0, we plan to promote the compile warning to a compiler error. We recommend that you migrate your code if you see this warning.

Documentation updates

The Kotlin documentation has received some notable changes:

- Get started with Spring Boot and Kotlin – create a simple application with a database and learn more about the features of Spring Boot and Kotlin.
- Scope functions – learn how to simplify your code with useful scope functions from the standard library.
- CocoaPods integration – set up an environment to work with CocoaPods.

Install Kotlin 1.8.20

Check the IDE version

IntelliJ IDEA 2022.2 and 2022.3 automatically suggest updating the Kotlin plugin to version 1.8.20. IntelliJ IDEA 2023.1 has the built-in Kotlin plugin 1.8.20.

Android Studio Flamingo (222) and Giraffe (223) will support Kotlin 1.8.20 in the next releases.

The new command-line compiler is available for download on the GitHub release page.

Configure Gradle settings

To download Kotlin artifacts and dependencies properly, update your settings.gradle(.kts) file to use the Maven Central repository:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

If the repository is not specified, Gradle uses the sunset JCenter repository that could lead to issues with Kotlin artifacts.

What's new in Kotlin 1.9.0-RC

Released: June 20, 2023

> This document doesn't cover all of the features of the Early Access Preview (EAP) release, but it highlights the latest ones and some major improvements.
>
> See the full list of changes in the GitHub changelog.

The Kotlin 1.9.0-RC release is out! Here are some highlights from this preview version of Kotlin:

- New Kotlin K2 compiler updates
- Stable replacement of the enum class values function
- Stable ..< operator for open-ended ranges
- New common function to get regex capture group by name
- New path utility to create parent directories
- Preview of Gradle configuration cache in Kotlin Multiplatform
- Changes for Android target support in Kotlin Multiplatform
- No object initialization when accessing constant values in Kotlin/Native
- Ability to configure standalone mode for iOS simulator tests in Kotlin/Native

IDE support

The Kotlin plugins that support 1.9.0-RC are available for:

IDE	Supported versions
IntelliJ IDEA	2022.3.x, 2023.1.x
Android Studio	Giraffe (223), Hedgehog (231)

New Kotlin K2 compiler updates

The Kotlin team continues to stabilize the K2 compiler. The 1.9.0-RC release introduces further advancements, including basic support for Kotlin/Native and improved Kotlin/JS stability in the K2 compiler. It's an important step towards full support of multiplatform projects. We would appreciate your feedback to help us with it.

Also, starting with 1.9.0-RC and until the release of Kotlin 2.0, you can easily test the K2 compiler in your projects. Add kotlin.experimental.tryK2=true to your gradle.properties file or run the following command:

```
./gradlew assemble -Pkotlin.experimental.tryK2=true
```

This Gradle property automatically sets the language version to 2.0 and updates the build report with the number of Kotlin tasks compiled using the K2 compiler compared to the current compiler:

```
##### 'kotlin.experimental.tryK2' results (Kotlin/Native not checked) #####
:lib:compileKotlin: 2.0 language version
:app:compileKotlin: 2.0 language version
##### 100% (2/2) tasks have been compiled with Kotlin 2.0 #####
```

Share your feedback on the new K2 compiler

We'd appreciate any feedback you might have!

- Provide your feedback directly to K2 developers in the Kotlin Slack – get an invite and join the #k2-early-adopters channel.
- Report any problems you've faced with the new K2 compiler via our issue tracker.
- Enable the Send usage statistics option to allow JetBrains to collect anonymous data about K2 usage.

Stable replacement of the enum class values function

In 1.8.20, the entries property for enum classes was introduced as an Experimental feature. The entries property is intended to be a modern and performant replacement for the synthetic values() function. In 1.9.0-RC, the entries property is Stable.

> The values() function is still supported, but we recommend that you use the entries property instead.

```
enum class Color(val colorName: String, val rgb: String) {
    RED("Red", "#FF0000"),
    ORANGE("Orange", "#FF7F00"),
    YELLOW("Yellow", "#FFFF00")
}

fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb == rgb }
```

For more information about the entries property for enum classes, see What's new in Kotlin 1.8.20.

Stable ..< operator for open-ended ranges

The new ..< operator for open-ended ranges that was introduced in Kotlin 1.7.20 is Stable in 1.9.0-RC. The standard library API for working with open-ended ranges is also Stable in this release.

Our research shows that the new ..< operator makes it easier to understand when an open-ended range is declared. If you use the until infix function, it's easy to make the mistake of assuming that the upper bound is included.

Here is an example using the until function:

```
fun main() {
    for (number in 2 until 10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 2 4 6 8
}
```

And here is an example using the new ..< operator:

```
fun main() {
    for (number in 2..<10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 2 4 6 8
}
```

> Starting with version 2023.1.1, IntelliJ IDEA has a new code inspection that highlights when you can use the ..< operator.

For more information about what you can do with this operator, see What's new in Kotlin 1.7.20.

New common function to get regex capture group by name

Prior to 1.9.0-RC, every platform had its own extension to get a regular expression capture group by its name from a regular expression match. However, there was no common function. It wasn't possible to implement prior to Kotlin 1.8.0, because the standard library still supported JVM targets 1.6 and 1.7.

As of Kotlin 1.8.0, the standard library is compiled with JVM target 1.8. So in 1.9.0-RC, there is now a common function groups that retrieves group's contents by its name for a regular expression match. This is useful when you want to access the results of regular expression matches belonging to a particular capture group.

Here is an example with a regular expression containing three capture groups: city, state, and areaCode. You can use these group names to access the matched values:

```
fun main() {
    val regex = """\b(?<city>[A-Za-z\s]+),\s(?<state>[A-Z]{2}):\s(?<areaCode>[0-9]{3})\b""".toRegex()
    val input = "Coordinates: Austin, TX: 123"


    val match = regex.find(input)!!
    println(match.groups["city"]?.value)
    // Austin
    println(match.groups["state"]?.value)
    // TX
    println(match.groups["areaCode"]?.value)
    // 123
}
```

New path utility to create parent directories

In 1.9.0-RC there is a new extension function createParentDirectories() that you can use to create a new file with all the necessary parent directories. When you provide a file path to createParentDirectories(), it checks whether the parent directories already exist. If they do, it does nothing. However, if they do not, it creates them for you.

createParentDirectories() is particularly useful when you are copying files. For example, you can use it in combination with the copyToRecursively() function:

```
sourcePath.copyToRecursively(
    destinationPath.createParentDirectories(),
    followLinks = false
)
```

Preview of Gradle configuration cache in Kotlin Multiplatform

Kotlin 1.9.0-RC comes with support for Gradle configuration cache in multiplatform libraries. If you're a library author, you can already benefit from the improved build performance.

Gradle configuration cache speeds up the build process by reusing the results of the configuration phase for subsequent builds. The feature has become Stable since Gradle 8.1. To enable it, follow the instructions in the Gradle documentation.

> The Kotlin Multiplatform plugin still doesn't support Gradle configuration cache with Xcode integration tasks or the Kotlin CocoaPods Gradle plugin. We expect to add this feature in a future Kotlin release.

Changes for Android target support in Kotlin Multiplatform

We continue our efforts to stabilize Kotlin Multiplatform. An essential step in this direction is to provide first-class support for the Android target. We're excited to announce that in the future, the Android team from Google will provide its own Gradle plugin to support Android in Kotlin Multiplatform.

To open the way for the new solution from Google, we're renaming the android block to androidTarget in the current Kotlin DSL in 1.9.0-RC. This is a temporary change that is necessary to free the android name for the upcoming DSL from Google.

The Google plugin will be the preferred way of working with Android in multiplatform projects. When it's ready, we'll provide the necessary migration instructions so that you'll be able to use the short android name as before.

No object initialization when accessing constant values in Kotlin/Native

Starting with Kotlin 1.9.0-RC, the Kotlin/Native backend doesn't initialize objects when accessing const val fields:

```
object MyObject {
    init {
        println("side effect!")
    }

    const val y = 1
}


fun main() {
    println(MyObject.y) // no initialization at first
    val x = MyObject // initialization occurs
    println(x.y)
}
```

Now the behavior is unified with Kotlin/JVM, where the implementation is consistent with Java and objects are never initialized in this case. You can also expect some performance improvements in your Kotlin/Native projects thanks to this change.

Ability to configure standalone mode for iOS simulator tests in Kotlin/Native

By default, when running iOS simulator tests for Kotlin/Native, the --standalone flag is used to avoid manual simulator booting and shutdown. In 1.9.0-RC, you can now configure whether this flag is used in a Gradle task via the standalone property. By default, the --standalone flag is used so standalone mode is enabled.

Here is an example of how to disable standalone mode in your build.gradle.kts file:

```
tasks.withType<org.jetbrains.kotlin.gradle.targets.native.tasks.KotlinNativeSimulatorTest>().configureEach {
    standalone.set(false)
}
```

> If you disable standalone mode, you must boot the simulator manually. To boot your simulator from CLI, you can use the following command:
>
> ```
> /usr/bin/xcrun simctl boot <DeviceId>
> ```

How to update to Kotlin 1.9.0-RC

Install Kotlin 1.9.0-RC in any of the following ways:

- If you use the Early Access Preview update channel, the IDE will suggest automatically updating to 1.9.0-RC as soon as it becomes available.
- If you use the Stable update channel, you can change the channel to Early Access Preview at any time by selecting Tools | Kotlin | Configure Kotlin Plugin Updates in your IDE. You'll then be able to install the latest preview release. Check out these instructions for details.

Once you've installed 1.9.0-RC don't forget to change the Kotlin version to 1.9.0-RC in your build scripts.

What's new in Kotlin 1.8.0

Released: 28 December 2022

The Kotlin 1.8.0 release is out and here are some of its biggest highlights:

- New experimental functions for JVM: recursively copy or delete directory content
- Improved kotlin-reflect performance
- New -Xdebug compiler option for better debugging experience
- kotlin-stdlib-jdk7 and kotlin-stdlib-jdk8 merged into kotlin-stdlib
- Improved Objective-C/Swift interoperability
- Compatibility with Gradle 7.3

IDE support

The Kotlin plugin that supports 1.8.0 is available for:

IDE	Supported versions
IntelliJ IDEA	2021.3, 2022.1, 2022.2
Android Studio	Electric Eel (221), Flamingo (222)

> You can update your projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3 without updating the IDE plugin.
>
> To migrate existing projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3, change the Kotlin version to 1.8.0 and reimport your Gradle or Maven project.

Kotlin/JVM

Starting with version 1.8.0, the compiler can generate classes with a bytecode version corresponding to JVM 19. The new language version also includes:

- A compiler option for switching off the generation of JVM annotation targets

- A new -Xdebug compiler option for disabling optimizations
- The removal of the old backend
- Support for Lombok's @Builder annotation

Ability to not generate TYPE_USE and TYPE_PARAMETER annotation targets

If a Kotlin annotation has TYPE among its Kotlin targets, the annotation maps to java.lang.annotation.ElementType.TYPE_USE in its list of Java annotation targets. This is just like how the TYPE_PARAMETER Kotlin target maps to the java.lang.annotation.ElementType.TYPE_PARAMETER Java target. This is an issue for Android clients with API levels less than 26, which don't have these targets in the API.

Starting with Kotlin 1.8.0, you can use the new compiler option -Xno-new-java-annotation-targets to avoid generating the TYPE_USE and TYPE_PARAMETER annotation targets.

A new compiler option for disabling optimizations

Kotlin 1.8.0 adds a new -Xdebug compiler option, which disables optimizations for a better debugging experience. For now, the option disables the "was optimized out" feature for coroutines. In the future, after we add more optimizations, this option will disable them, too.

The "was optimized out" feature optimizes variables when you use suspend functions. However, it is difficult to debug code with optimized variables because you don't see their values.

> Never use this option in production: Disabling this feature via -Xdebug can cause memory leaks.

Removal of the old backend

In Kotlin 1.5.0, we announced that the IR-based backend became Stable. That meant that the old backend from Kotlin 1.4.* was deprecated. In Kotlin 1.8.0, we've removed the old backend completely. By extension, we've removed the compiler option -Xuse-old-backend and the Gradle useOldBackend option.

Support for Lombok's @Builder annotation

The community has added so many votes for the Kotlin Lombok: Support generated builders (@Builder) YouTrack issue that we just had to support the @Builder annotation.

We don't yet have plans to support the @SuperBuilder or @Tolerate annotations, but we'll reconsider if enough people vote for the @SuperBuilder and @Tolerate issues.

Learn how to configure the Lombok compiler plugin.

Kotlin/Native

Kotlin 1.8.0 includes changes to Objective-C and Swift interoperability, support for Xcode 14.1, and improvements to the CocoaPods Gradle plugin:

- Support for Xcode 14.1
- Improved Objective-C/Swift interoperability
- Dynamic frameworks by default in the CocoaPods Gradle plugin

Support for Xcode 14.1

The Kotlin/Native compiler now supports the latest stable Xcode version, 14.1. The compatibility improvements include the following changes:

- There's a new watchosDeviceArm64 preset for the watchOS target that supports Apple watchOS on ARM64 platforms.
- The Kotlin CocoaPods Gradle plugin no longer has bitcode embedding for Apple frameworks by default.
- Platform libraries were updated to reflect the changes to Objective-C frameworks for Apple targets.

Improved Objective-C/Swift interoperability

To make Kotlin more interoperable with Objective-C and Swift, three new annotations were added:

- @ObjCName allows you to specify a more idiomatic name in Swift or Objective-C, instead of renaming the Kotlin declaration.

  The annotation instructs the Kotlin compiler to use a custom Objective-C and Swift name for this class, property, parameter, or function:

```
@ObjCName(swiftName = "MySwiftArray")
class MyKotlinArray {
    @ObjCName("index")
    fun indexOf(@ObjCName("of") element: String): Int = TODO()
}

// Usage with the ObjCName annotations
let array = MySwiftArray()
let index = array.index(of: "element")
```

- @HiddenFromObjC allows you to hide a Kotlin declaration from Objective-C.

  The annotation instructs the Kotlin compiler not to export a function or property to Objective-C and, consequently, Swift. This can make your Kotlin code more Objective-C/Swift-friendly.

- @ShouldRefineInSwift is useful for replacing a Kotlin declaration with a wrapper written in Swift.

  The annotation instructs the Kotlin compiler to mark a function or property as swift_private in the generated Objective-C API. Such declarations get the __ prefix, which makes them invisible to Swift code.

  You can still use these declarations in your Swift code to create a Swift-friendly API, but they won't be suggested by Xcode's autocompletion, for example.

  For more information on refining Objective-C declarations in Swift, see the official Apple documentation.

> The new annotations require opt-in.

The Kotlin team is very grateful to Rick Clephas for implementing these annotations.

Dynamic frameworks by default in the CocoaPods Gradle plugin

Starting with Kotlin 1.8.0, Kotlin frameworks registered by the CocoaPods Gradle plugin are linked dynamically by default. The previous static implementation was inconsistent with the behavior of the Kotlin Gradle plugin.

```
kotlin {
    cocoapods {
        framework {
            baseName = "MyFramework"
            isStatic = false // Now dynamic by default
        }
    }
}
```

If you have an existing project with a static linking type and you upgrade to Kotlin 1.8.0 (or change the linking type explicitly), you may encounter an error with the project's execution. To fix it, close your Xcode project and run pod install in the Podfile directory.

For more information, see the CocoaPods Gradle plugin DSL reference.

Kotlin Multiplatform: A new Android source set layout

Kotlin 1.8.0 introduces a new Android source set layout that replaces the previous naming schema for directories, which is confusing in multiple ways.

Consider an example of two androidTest directories created in the current layout. One is for KotlinSourceSets and the other is for AndroidSourceSets:

- They have different semantics: Kotlin's androidTest belongs to the unitTest type, whereas Android's belongs to the integrationTest type.
- They create a confusing SourceDirectories layout, as src/androidTest/java has a UnitTest and src/androidTest/kotlin has an InstrumentedTest.
- Both KotlinSourceSets and AndroidSourceSets use a similar naming schema for Gradle configurations, so the resulting configurations of androidTest for both Kotlin's and Android's source sets are the same: androidTestImplementation, androidTestApi, androidTestRuntimeOnly, and androidTestCompileOnly.

To address these and other existing issues, we've introduced a new Android source set layout. Here are some of the key differences between the two layouts:

KotlinSourceSet naming schema

Current source set layout	New source set layout
targetName + AndroidSourceSet.name	targetName + AndroidVariantType

{AndroidSourceSet.name} maps to {KotlinSourceSet.name} as follows:

	Current source set layout	New source set layout
main	androidMain	androidMain
test	androidTest	androidUnitTest
androidTest	androidAndroidTest	androidInstrumentedTest

SourceDirectories

Current source set layout	New source set layout
The layout adds additional /kotlin SourceDirectories	src/{AndroidSourceSet.name}/kotlin, src/{KotlinSourceSet.name}/kotlin

{AndroidSourceSet.name} maps to {SourceDirectories included} as follows:

	Current source set layout	New source set layout
main	src/androidMain/kotlin, src/main/kotlin, src/main/java	src/androidMain/kotlin, src/main/kotlin, src/main/java
test	src/androidTest/kotlin, src/test/kotlin, src/test/java	src/androidUnitTest/kotlin, src/test/kotlin, src/test/java
androidTest	src/androidAndroidTest/kotlin, src/androidTest/java	src/androidInstrumentedTest/kotlin, src/androidTest/java, src/androidTest/kotlin

The location of the AndroidManifest.xml file

Current source set layout	New source set layout
src/{AndroidSourceSet.name}/AndroidManifest.xml	src/{KotlinSourceSet.name}/AndroidManifest.xml

{AndroidSourceSet.name} maps to{AndroidManifest.xml location} as follows:

	Current source set layout	New source set layout
main	src/main/AndroidManifest.xml	src/androidMain/AndroidManifest.xml

	Current source set layout	New source set layout
debug	src/debug/AndroidManifest.xml	src/androidDebug/AndroidManifest.xml

The relation between Android and common tests

The new Android source set layout changes the relation between Android-instrumented tests (renamed to androidInstrumentedTest in the new layout) and common tests.

Previously, there was a default dependsOn relation between androidAndroidTest and commonTest. In practice, it meant the following:

- The code in commonTest was available in androidAndroidTest.
- expect declarations in commonTest had to have corresponding actual implementations in androidAndroidTest.
- Tests declared in commonTest were also running as Android instrumented tests.

In the new Android source set layout, the dependsOn relation is not added by default. If you prefer the previous behavior, manually declare this relation in your build.gradle.kts file:

```kotlin
kotlin {
// ...
    sourceSets {
        val commonTest by getting
        val androidInstrumentedTest by getting {
            dependsOn(commonTest)
        }
    }
}
```

Support for Android flavors

Previously, the Kotlin Gradle plugin eagerly created source sets that correspond to Android source sets with debug and release build types or custom flavors like demo and full. It made them accessible by constructions like val androidDebug by getting { ... }.

In the new Android source set layout, those source sets are created in the afterEvaluate phase. It makes such expressions invalid, leading to errors like org.gradle.api.UnknownDomainObjectException: KotlinSourceSet with name 'androidDebug' not found.

To work around that, use the new invokeWhenCreated() API in your build.gradle.kts file:

```kotlin
kotlin {
// ...
    sourceSets.invokeWhenCreated("androidFreeDebug") {
        // ...
    }
}
```

Configuration and setup

The new layout will become the default in future releases. You can enable it now with the following Gradle option:

```
kotlin.mpp.androidSourceSetLayoutVersion=2
```

> The new layout requires Android Gradle plugin 7.0 or later and is supported in Android Studio 2022.3 and later.

The usage of the previous Android-style directories is now discouraged. Kotlin 1.8.0 marks the start of the deprecation cycle, introducing a warning for the current layout. You can suppress the warning with the following Gradle property:

```
kotlin.mpp.androidSourceSetLayoutVersion1.nowarn=true
```

Kotlin/JS

Kotlin 1.8.0 stabilizes the JS IR compiler backend and brings new features to JavaScript-related Gradle build scripts:

- Stable JS IR compiler backend
- New settings for reporting that yarn.lock has been updated
- Add test targets for browsers via Gradle properties
- New approach to adding CSS support to your project

Stable JS IR compiler backend

Starting with this release, the Kotlin/JS intermediate representation (IR-based) compiler backend is Stable. It took a while to unify infrastructure for all three backends, but they now work with the same IR for Kotlin code.

As a consequence of the stable JS IR compiler backend, the old one is deprecated from now on.

Incremental compilation is enabled by default along with the stable JS IR compiler.

If you still use the old compiler, switch your project to the new backend with the help of our migration guide.

New settings for reporting that yarn.lock has been updated

If you use the yarn package manager, there are three new special Gradle settings that could notify you if the yarn.lock file has been updated. You can use these settings when you want to be notified if yarn.lock has been changed silently during the CI build process.

These three new Gradle properties are:

- YarnLockMismatchReport, which specifies how changes to the yarn.lock file are reported. You can use one of the following values:
  - FAIL fails the corresponding Gradle task. This is the default.
  - WARNING writes the information about changes in the warning log.
  - NONE disables reporting.
- reportNewYarnLock, which reports about the recently created yarn.lock file explicitly. By default, this option is disabled: it's a common practice to generate a new yarn.lock file at the first start. You can use this option to ensure that the file has been committed to your repository.
- yarnLockAutoReplace, which replaces yarn.lock automatically every time the Gradle task is run.

To use these options, update your build script file build.gradle.kts as follows:

```kotlin
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin::class.java) {
    rootProject.the<YarnRootExtension>().yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.the<YarnRootExtension>().reportNewYarnLock = false // true
    rootProject.the<YarnRootExtension>().yarnLockAutoReplace = false // true
}
```

Add test targets for browsers via Gradle properties

Starting with Kotlin 1.8.0, you can set test targets for different browsers right in the Gradle properties file. Doing so shrinks the size of the build script file as you no longer need to write all targets in build.gradle.kts.

You can use this property to define a list of browsers for all modules, and then add specific browsers in the build scripts of particular modules.

For example, the following line in your Gradle property file will run the test in Firefox and Safari for all modules:

```
kotlin.js.browser.karma.browsers=firefox,safari
```

See the full list of available values for the property on GitHub.

The Kotlin team is very grateful to Martynas Petuška for implementing this feature.

New approach to adding CSS support to your project

This release provides a new approach to adding CSS support to your project. We assume that this will affect a lot of projects, so don't forget to update your Gradle build script files as described below.

Before Kotlin 1.8.0, the cssSupport.enabled property was used to add CSS support:

```
browser {
    commonWebpackConfig {
        cssSupport.enabled = true
    }
}
```

Now you should use the enabled.set() method in the cssSupport {} block:

```
browser {
    commonWebpackConfig {
        cssSupport {
            enabled.set(true)
        }
    }
}
```

Gradle

Kotlin 1.8.0 fully supports Gradle versions 7.2 and 7.3. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings lots of changes:

- Exposing Kotlin compiler options as Gradle lazy properties
- Bumping the minimum supported versions
- Ability to disable the Kotlin daemon fallback strategy
- Usage of the latest kotlin-stdlib version in transitive dependencies
- Obligatory check for JVM target compatibility equality of related Kotlin and Java compile tasks
- Resolution of Kotlin Gradle plugins' transitive dependencies
- Deprecations and removals

Exposing Kotlin compiler options as Gradle lazy properties

To expose available Kotlin compiler options as Gradle lazy properties and to integrate them better into the Kotlin tasks, we made lots of changes:

- Compile tasks have the new compilerOptions input, which is similar to the existing kotlinOptions but uses Property from the Gradle Properties API as the return type:

  ```
  tasks.named("compileKotlin", org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile::class.java) {
      compilerOptions {
          useK2.set(true)
      }
  }
  ```

- The Kotlin tools tasks KotlinJsDce and KotlinNativeLink have the new toolOptions input, which is similar to the existing kotlinOptions input.
- New inputs have the @Nested Gradle annotation. Every property inside the inputs has a related Gradle annotation, such as @Input or @Internal.
- The Kotlin Gradle plugin API artifact has two new interfaces:
  - org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask, which has the compilerOptions input and the compileOptions() method. All Kotlin compilation tasks implement this interface.

- org.jetbrains.kotlin.gradle.tasks.KotlinToolTask, which has the toolOptions input and the toolOptions() method. All Kotlin tool tasks – KotlinJsDce, KotlinNativeLink, and KotlinNativeLinkArtifactTask – implement this interface.

- Some compilerOptions use the new types instead of the String type:
  - JvmTarget
  - KotlinVersion (for the apiVersion and the languageVersion inputs)
  - JsMainFunctionExecutionMode
  - JsModuleKind
  - JsSourceMapEmbedMode

  For example, you can use compilerOptions.jvmTarget.set(JvmTarget.JVM_11) instead of kotlinOptions.jvmTarget = "11".

  The kotlinOptions types didn't change, and they are internally converted to compilerOptions types.

- The Kotlin Gradle plugin API is binary-compatible with previous releases. There are, however, some source and ABI-breaking changes in the kotlin-gradle-plugin artifact. Most of these changes involve additional generic parameters to some internal types. One important change is that the KotlinNativeLink task no longer inherits the AbstractKotlinNativeCompile task.

- KotlinJsCompilerOptions.outputFile and the related KotlinJsOptions.outputFile options are deprecated. Use the Kotlin2JsCompile.outputFileProperty task input instead.

> The Kotlin Gradle plugin still adds the KotlinJvmOptions DSL to the Android extension:
>
> ```
> android {
>     kotlinOptions {
>         jvmTarget = "11"
>     }
> }
> ```
>
> This will be changed in the scope of this issue, when the compilerOptions DSL will be added to a module level.

Limitations

> The kotlinOptions task input and the kotlinOptions{...} task DSL are in support mode and will be deprecated in upcoming releases. Improvements will be made only to compilerOptions and toolOptions.

Calling any setter or getter on kotlinOptions delegates to the related property in the compilerOptions. This introduces the following limitations:

- compilerOptions and kotlinOptions cannot be changed in the task execution phase (see one exception in the paragraph below).
- freeCompilerArgs returns an immutable List<String>, which means that, for example, kotlinOptions.freeCompilerArgs.remove("something") will fail.

Several plugins, including kotlin-dsl and the Android Gradle plugin (AGP) with Jetpack Compose enabled, try to modify the freeCompilerArgs attribute in the task execution phase. We've added a workaround for them in Kotlin 1.8.0. This workaround allows any build script or plugin to modify kotlinOptions.freeCompilerArgs in the execution phase but produces a warning in the build log. To disable this warning, use the new Gradle property kotlin.options.suppressFreeCompilerArgsModificationWarning=true. Gradle is going to add fixes for the kotlin-dsl plugin and AGP with Jetpack Compose enabled.

Bumping the minimum supported versions

Starting with Kotlin 1.8.0, the minimum supported Gradle version is 6.8.3 and the minimum supported Android Gradle plugin version is 4.1.3.

See the Kotlin Gradle plugin compatibility with available Gradle versions in our documentation

Ability to disable the Kotlin daemon fallback strategy

There is a new Gradle property kotlin.daemon.useFallbackStrategy, whose default value is true. When the value is false, builds fail on problems with the daemon's startup or communication. There is also a new useDaemonFallbackStrategy property in Kotlin compile tasks, which takes priority over the Gradle property if you use both. If there is insufficient memory to run the compilation, you can see a message about it in the logs.

The Kotlin compiler's fallback strategy is to run a compilation outside the Kotlin daemon if the daemon somehow fails. If the Gradle daemon is on, the compiler uses the "In process" strategy. If the Gradle daemon is off, the compiler uses the "Out of process" strategy. Learn more about these execution strategies in the documentation. Note that silent fallback to another strategy can consume a lot of system resources or lead to non-deterministic builds; see this YouTrack issue for more details.

Usage of the latest kotlin-stdlib version in transitive dependencies

If you explicitly write Kotlin version 1.8.0 or higher in your dependencies, for example: implementation("org.jetbrains.kotlin:kotlin-stdlib:1.8.0"), then the Kotlin Gradle Plugin will use that Kotlin version for transitive kotlin-stdlib-jdk7 and kotlin-stdlib-jdk8 dependencies. This is done to avoid class duplication from different stdlib versions (learn more about merging kotlin-stdlib-jdk7 and kotlin-stdlib-jdk8 into kotlin-stdlib). You can disable this behavior with the kotlin.stdlib.jdk.variants.version.alignment Gradle property:

```
kotlin.stdlib.jdk.variants.version.alignment=false
```

If you run into issues with version alignment, align all versions via the Kotlin BOM by declaring a platform dependency on kotlin-bom in your build script:

```
implementation(platform("org.jetbrains.kotlin:kotlin-bom:1.8.0"))
```

Learn about other cases and our suggested solutions in the documentation.

Obligatory check for JVM targets of related Kotlin and Java compile tasks

> This section applies to your JVM project even if your source files are only in Kotlin and you don't use Java.

Starting from this release, the default value for the kotlin.jvm.target.validation.mode property is error for projects on Gradle 8.0+ (this version of Gradle has not been released yet), and the plugin will fail the build in the event of JVM target incompatibility.

The shift of the default value from warning to error is a preparation step for a smooth migration to Gradle 8.0. We encourage you to set this property to error and configure a toolchain or align JVM versions manually.

Learn more about what can go wrong if you don't check the targets' compatibility.

Resolution of Kotlin Gradle plugins' transitive dependencies

In Kotlin 1.7.0, we introduced support for Gradle plugin variants. Because of these plugin variants, a build classpath can have different versions of the Kotlin Gradle plugins that depend on different versions of some dependency, usually kotlin-gradle-plugin-api. This can lead to a resolution problem, and we would like to propose the following workaround, using the kotlin-dsl plugin as an example.

The kotlin-dsl plugin in Gradle 7.6 depends on the org.jetbrains.kotlin.plugin.sam.with.receiver:1.7.10 plugin, which depends on kotlin-gradle-plugin-api:1.7.10. If you add the org.jetbrains.kotlin.gradle.jvm:1.8.0 plugin, this kotlin-gradle-plugin-api:1.7.10 transitive dependency may lead to a dependency resolution error because of a mismatch between the versions (1.8.0 and 1.7.10) and the variant attributes' org.gradle.plugin.api-version values. As a workaround, add this constraint to align the versions. This workaround may be needed until we implement the Kotlin Gradle Plugin libraries alignment platform, which is in the plans:

```
dependencies {
    constraints {
        implementation("org.jetbrains.kotlin:kotlin-sam-with-receiver:1.8.0")
    }
}
```

This constraint forces the org.jetbrains.kotlin:kotlin-sam-with-receiver:1.8.0 version to be used in the build classpath for transitive dependencies. Learn more about one similar case in the Gradle issue tracker.

Deprecations and removals

In Kotlin 1.8.0, the deprecation cycle continues for the following properties and methods:

- In the notes for Kotlin 1.7.0 that the KotlinCompile task still had the deprecated Kotlin property classpath, which would be removed in future releases. Now, we've changed the deprecation level to error for the KotlinCompile task's classpath property. All compile tasks use the libraries input for a list of libraries required for compilation.

- We removed the kapt.use.worker.api property that allowed running kapt via the Gradle Workers API. By default, kapt has been using Gradle workers since Kotlin 1.3.70, and we recommend sticking to this method.
- In Kotlin 1.7.0, we announced the start of a deprecation cycle for the kotlin.compiler.execution.strategy property. In this release, we removed this property. Learn how to define a Kotlin compiler execution strategy in other ways.

Standard library

Kotlin 1.8.0:

- Updates JVM compilation target.
- Stabilizes a number of functions – TimeUnit conversion between Java and Kotlin, cbrt(), Java Optionals extension functions.
- Provides a preview for comparable and subtractable TimeMarks.
- Includes experimental extension functions for java.nio.file.path.
- Presents improved kotlin-reflect performance.

Updated JVM compilation target

In Kotlin 1.8.0, the standard libraries (kotlin-stdlib, kotlin-reflect, and kotlin-script-*) are compiled with JVM target 1.8. Previously, the standard libraries were compiled with JVM target 1.6.

Kotlin 1.8.0 no longer supports JVM targets 1.6 and 1.7. As a result, you no longer need to declare kotlin-stdlib-jdk7 and kotlin-stdlib-jdk8 separately in build scripts because the contents of these artifacts have been merged into kotlin-stdlib.

> If you have explicitly declared kotlin-stdlib-jdk7 and kotlin-stdlib-jdk8 as dependencies in your build scripts, then you should replace them with kotlin-stdlib.

Note that mixing different versions of stdlib artifacts could lead to class duplication or to missing classes. To avoid that, the Kotlin Gradle plugin can help you align stdlib versions.

cbrt()

The cbrt() function, which allows you to compute the real cube root of a double or float, is now Stable.

```
import kotlin.math.*

fun main() {
    val num = 27
    val negNum = -num

    println("The cube root of ${num.toDouble()} is: " +
            cbrt(num.toDouble()))
    println("The cube root of ${negNum.toDouble()} is: " +
            cbrt(negNum.toDouble()))
}
```

TimeUnit conversion between Java and Kotlin

The toTimeUnit() and toDurationUnit() functions in kotlin.time are now Stable. Introduced as Experimental in Kotlin 1.6.0, these functions improve interoperability between Kotlin and Java. You can now easily convert between Java java.util.concurrent.TimeUnit and Kotlin kotlin.time.DurationUnit. These functions are supported on the JVM only.

```
import kotlin.time.*

// For use from Java
fun wait(timeout: Long, unit: TimeUnit) {
    val duration: Duration = timeout.toDuration(unit.toDurationUnit())
    ...
}
```

Comparable and subtractable TimeMarks

The new functionality of TimeMarks is Experimental, and to use it you need to opt in by using @OptIn(ExperimentalTime::class) or @ExperimentalTime.

Before Kotlin 1.8.0, if you wanted to calculate the time difference between multiple TimeMarks and now, you could only call elapsedNow() on one TimeMark at a time. This made it difficult to compare the results because the two elapsedNow() function calls couldn't be executed at exactly the same time.

To solve this, in Kotlin 1.8.0 you can subtract and compare TimeMarks from the same time source. Now you can create a new TimeMark instance to represent now and subtract other TimeMarks from it. This way, the results that you collect from these calculations are guaranteed to be relative to each other.

```kotlin
import kotlin.time.*
fun main() {
//sampleStart
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    Thread.sleep(500) // Sleep 0.5 seconds
    val mark2 = timeSource.markNow()

    // Before 1.8.0
    repeat(4) { n ->
        val elapsed1 = mark1.elapsedNow()
        val elapsed2 = mark2.elapsedNow()

        // Difference between elapsed1 and elapsed2 can vary depending
        // on how much time passes between the two elapsedNow() calls
        println("Measurement 1.${n + 1}: elapsed1=$elapsed1, " +
                "elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    println()

    // Since 1.8.0
    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        // Now the elapsed times are calculated relative to mark3,
        // which is a fixed value
        println("Measurement 2.${n + 1}: elapsed1=$elapsed1, " +
                "elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    // It's also possible to compare time marks with each other
    // This is true, as mark2 was captured later than mark1
    println(mark2 > mark1)
//sampleEnd
}
```

This new functionality is particularly useful in animation calculations where you want to calculate the difference between, or compare, multiple TimeMarks representing different frames.

Recursive copying or deletion of directories

These new functions for java.nio.file.path are Experimental. To use them, you need to opt in with @OptIn(kotlin.io.path.ExperimentalPathApi::class) or @kotlin.io.path.ExperimentalPathApi. Alternatively, you can use the compiler option -opt-in=kotlin.io.path.ExperimentalPathApi.

We have introduced two new extension functions for java.nio.file.Path, copyToRecursively() and deleteRecursively(), which allow you to recursively:

- Copy a directory and its contents to another destination.
- Delete a directory and its contents.

These functions can be very useful as part of a backup process.

Error handling

Using copyToRecursively(), you can define what should happen if an exception occurs while copying, by overloading the onError lambda function:

```kotlin
sourceRoot.copyToRecursively(destinationRoot, followLinks = false,
    onError = { source, target, exception ->
        logger.logError(exception, "Failed to copy $source to $target")
        OnErrorResult.TERMINATE
    })
```

When you use deleteRecursively(), if an exception occurs while deleting a file or folder, then the file or folder is skipped. Once the deletion has completed, deleteRecursively() throws an IOException containing all the exceptions that occurred as suppressed exceptions.

File overwrite

If copyToRecursively() finds that a file already exists in the destination directory, then an exception occurs. If you want to overwrite the file instead, use the overload that has overwrite as an argument and set it to true:

```kotlin
fun setUpEnvironment(projectDirectory: Path, fixtureName: String) {
    fixturesRoot.resolve(COMMON_FIXTURE_NAME)
        .copyToRecursively(projectDirectory, followLinks = false)
    fixturesRoot.resolve(fixtureName)
        .copyToRecursively(projectDirectory, followLinks = false,
            overwrite = true) // patches the common fixture
}
```

Custom copying action

To define your own custom logic for copying, use the overload that has copyAction as an additional argument. By using copyAction you can provide a lambda function, for example, with your preferred actions:

```kotlin
sourceRoot.copyToRecursively(destinationRoot, followLinks = false) { source, target ->
    if (source.name.startsWith(".")) {
        CopyActionResult.SKIP_SUBTREE
    } else {
        source.copyToIgnoringExistingDirectory(target, followLinks = false)
        CopyActionResult.CONTINUE
    }
}
```

For more information on these extension functions, see our API reference.

Java Optionals extension functions

The extension functions that were introduced in Kotlin 1.7.0 are now Stable. These functions simplify working with Optional classes in Java. They can be used to unwrap and convert Optional objects on the JVM, and to make working with Java APIs more concise. For more information, see What's new in Kotlin 1.7.0.

Improved kotlin-reflect performance

Taking advantage of the fact that kotlin-reflect is now compiled with JVM target 1.8, we migrated our internal cache mechanism to Java's ClassValue. Previously we only cached KClass, but we now also cache KType and KDeclarationContainer. These changes have led to significant performance improvements when invoking typeOf().

Documentation updates

The Kotlin documentation has received some notable changes:

Revamped and new pages

- Gradle overview – learn how to configure and build a Kotlin project with the Gradle build system, available compiler options, compilation, and caches in the Kotlin Gradle plugin.
- Nullability in Java and Kotlin – see the differences between Java's and Kotlin's approaches to handling possibly nullable variables.
- Lincheck guide – learn how to set up and use the Lincheck framework for testing concurrent algorithms on the JVM.

New and updated tutorials

- Get started with Gradle and Kotlin/JVM – create a console application using IntelliJ IDEA and Gradle.
- Create a multiplatform app using Ktor and SQLDelight – create a mobile application for iOS and Android using Kotlin Multiplatform Mobile.
- Get started with Kotlin Multiplatform Mobile – learn about cross-platform mobile development with Kotlin and create an app that works on both Android and iOS.

Install Kotlin 1.8.0

IntelliJ IDEA 2021.3, 2022.1, and 2022.2 automatically suggest updating the Kotlin plugin to version 1.8.0. IntelliJ IDEA 2022.3 will have the 1.8.0 version of the Kotlin plugin bundled in an upcoming minor update.

> To migrate existing projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3, change the Kotlin version to 1.8.0 and reimport your Gradle or Maven project.

For Android Studio Electric Eel (221) and Flamingo (222), version 1.8.0 of the Kotlin plugin will be delivered with the upcoming Android Studios updates. The new command-line compiler is available for download on the GitHub release page.

Compatibility guide for Kotlin 1.8.0

Kotlin 1.8.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of these changes in the Compatibility guide for Kotlin 1.8.0.

What's new in Kotlin 1.7.20

Released: 29 September 2022

The Kotlin 1.7.20 release is out! Here are some highlights from this release:

- The new Kotlin K2 compiler supports all-open, SAM with receiver, Lombok, and other compiler plugins
- We introduced the preview of the ..< operator for creating open-ended ranges
- The new Kotlin/Native memory manager is now enabled by default
- We introduced a new experimental feature for JVM: inline classes with a generic underlying type

You can also find a short overview of the changes in this video:

Watch video online.

Support for Kotlin K2 compiler plugins

The Kotlin team continues to stabilize the K2 compiler. K2 is still in Alpha (as announced in the Kotlin 1.7.0 release), but it now supports several compiler plugins. You can follow this YouTrack issue to get updates from the Kotlin team on the new compiler.

Starting with this 1.7.20 release, the Kotlin K2 compiler supports the following plugins:

- all-open
- no-arg
- SAM with receiver
- Lombok
- AtomicFU
- jvm-abi-gen

> The Alpha version of the new K2 compiler only works with JVM projects. It doesn't support Kotlin/JS, Kotlin/Native, or other multiplatform projects.

Learn more about the new compiler and its benefits in the following videos:

- The Road to the New Kotlin Compiler
- K2 Compiler: a Top-Down View

How to enable the Kotlin K2 compiler

To enable the Kotlin K2 compiler and test it, use the following compiler option:

```
-Xuse-k2
```

You can specify it in your build.gradle(.kts) file:

Kotlin

```kotlin
tasks.withType<KotlinCompile> {
    kotlinOptions.useK2 = true
}
```

Groovy

```groovy
compileKotlin {
    kotlinOptions.useK2 = true
}
```

Check out the performance boost on your JVM projects and compare it with the results of the old compiler.

Leave your feedback on the new K2 compiler

We really appreciate your feedback in any form:

- Provide your feedback directly to K2 developers in Kotlin Slack: get an invite and join the #k2-early-adopters channel.
- Report any problems you faced with the new K2 compiler to our issue tracker.
- Enable the Send usage statistics option to allow JetBrains collecting anonymous data about K2 usage.

Language

Kotlin 1.7.20 introduces preview versions for new language features, as well as puts restrictions on builder type inference:

- Preview of the ..< operator for creating open-ended ranges
- New data object declarations
- Builder type inference restrictions

Preview of the ..< operator for creating open-ended ranges

> The new operator is Experimental, and it has limited support in the IDE.

This release introduces the new ..< operator. Kotlin has the .. operator to express a range of values. The new ..< operator acts like the until function and helps you define the open-ended range.

Watch video online.

Our research shows that this new operator does a better job at expressing open-ended ranges and making it clear that the upper bound is not included.

Here is an example of using the ..< operator in a when expression:

```
when (value) {
    in 0.0..<0.25 -> // First quarter
    in 0.25..<0.5 -> // Second quarter
    in 0.5..<0.75 -> // Third quarter
    in 0.75..1.0 ->  // Last quarter  <- Note closed range here
}
```

Standard library API changes

The following new types and operations will be introduced in the kotlin.ranges packages in the common Kotlin standard library:

New OpenEndRange interface

The new interface to represent open-ended ranges is very similar to the existing ClosedRange<T> interface:

```
interface OpenEndRange<T : Comparable<T>> {
    // Lower bound
    val start: T
    // Upper bound, not included in the range
    val endExclusive: T
    operator fun contains(value: T): Boolean = value >= start && value < endExclusive
    fun isEmpty(): Boolean = start >= endExclusive
}
```

Implementing OpenEndRange in the existing iterable ranges

When developers need to get a range with an excluded upper bound, they currently use the until function to effectively produce a closed iterable range with the same values. To make these ranges acceptable in the new API that takes OpenEndRange<T>, we want to implement that interface in the existing iterable ranges: IntRange, LongRange, CharRange, UIntRange, and ULongRange. So they will simultaneously implement both the ClosedRange<T> and OpenEndRange<T> interfaces.

```
class IntRange : IntProgression(...), ClosedRange<Int>, OpenEndRange<Int> {
    override val start: Int
    override val endInclusive: Int
    override val endExclusive: Int
}
```

rangeUntil operators for the standard types

The rangeUntil operators will be provided for the same types and combinations currently defined by the rangeTo operator. We provide them as extension functions for prototype purposes, but for consistency, we plan to make them members later before stabilizing the open-ended ranges API.

How to enable the ..< operator

To use the ..< operator or to implement that operator convention for your own types, enable the -language-version 1.8 compiler option.

The new API elements introduced to support the open-ended ranges of the standard types require an opt-in, as usual for an experimental stdlib API: @OptIn(ExperimentalStdlibApi::class). Alternatively, you could use the -opt-in=kotlin.ExperimentalStdlibApi compiler option.

Read more about the new operator in this KEEP document.

Improved string representations for singletons and sealed class hierarchies with data objects

> Data objects are Experimental, and have limited support in the IDE at the moment.

This release introduces a new type of object declaration for you to use: data object. Data object behaves conceptually identical to a regular object declaration but comes with a clean toString representation out of the box.

Watch video online.

```
package org.example
object MyObject
data object MyDataObject

fun main() {
    println(MyObject) // org.example.MyObject@1f32e575
    println(MyDataObject) // MyDataObject
}
```

This makes data object declarations perfect for sealed class hierarchies, where you may use them alongside data class declarations. In this snippet, declaring EndOfFile as a data object instead of a plain object means that it will get a pretty toString without the need to override it manually, maintaining symmetry with the

accompanying data class definitions:

```
sealed class ReadResult {
    data class Number(val value: Int) : ReadResult()
    data class Text(val value: String) : ReadResult()
    data object EndOfFile : ReadResult()
}

fun main() {
    println(ReadResult.Number(1)) // Number(value=1)
    println(ReadResult.Text("Foo")) // Text(value=Foo)
    println(ReadResult.EndOfFile) // EndOfFile
}
```

How to enable data objects

To use data object declarations in your code, enable the -language-version 1.9 compiler option. In a Gradle project, you can do so by adding the following to your build.gradle(.kts):

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile>().configureEach {
    // ...
    kotlinOptions.languageVersion = "1.9"
}
```

Groovy

```
compileKotlin {
    // ...
    kotlinOptions.languageVersion = '1.9'
}
```

Read more about data objects, and share your feedback on their implementation in the respective KEEP document.

New builder type inference restrictions

Kotlin 1.7.20 places some major restrictions on the use of builder type inference that could affect your code. These restrictions apply to code containing builder lambda functions, where it's impossible to derive the parameter without analyzing the lambda itself. The parameter is used as an argument. Now, the compiler will always show an error for such code and ask you to specify the type explicitly.

This is a breaking change, but our research shows that these cases are very rare, and the restrictions shouldn't affect your code. If they do, consider the following cases:

- Builder inference with extension that hides members.

  If your code contains an extension function with the same name that will be used during the builder inference, the compiler will show you an error:

  ```
  class Data {
      fun doSmth() {} // 1
  }

  fun <T> T.doSmth() {} // 2

  fun test() {
      buildList {
          this.add(Data())
          this.get(0).doSmth() // Resolves to 2 and leads to error
      }
  }
  ```

  To fix the code, you should specify the type explicitly:

  ```
  class Data {
      fun doSmth() {} // 1
  }
  ```

```kotlin
fun <T> T.doSmth() {} // 2

fun test() {
    buildList<Data> { // Type argument!
        this.add(Data())
        this.get(0).doSmth() // Resolves to 1
    }
}
```

- Builder inference with multiple lambdas and the type arguments are not specified explicitly.

  If there are two or more lambda blocks in builder inference, they affect the type. To prevent an error, the compiler requires you to specify the type:

```kotlin
fun <T: Any> buildList(
    first: MutableList<T>.() -> Unit,
    second: MutableList<T>.() -> Unit
): List<T> {
    val list = mutableListOf<T>()
    list.first()
    list.second()
    return list
}

fun main() {
    buildList(
        first = { // this: MutableList<String>
            add("")
        },
        second = { // this: MutableList<Int>
            val i: Int = get(0)
            println(i)
        }
    )
}
```

  To fix the error, you should specify the type explicitly and fix the type mismatch:

```kotlin
fun main() {
    buildList<Int>(
        first = { // this: MutableList<Int>
            add(0)
        },
        second = { // this: MutableList<Int>
            val i: Int = get(0)
            println(i)
        }
    )
}
```

If you haven't found your case mentioned above, file an issue to our team.

See this YouTrack issue for more information about this builder inference update.

Kotlin/JVM

Kotlin 1.7.20 introduces generic inline classes, adds more bytecode optimizations for delegated properties, and supports IR in the kapt stub generating task, making it possible to use all the newest Kotlin features with kapt:

- Generic inline classes
- More optimized cases of delegated properties
- Support for the JVM IR backend in kapt stub generating task

Generic inline classes

> Generic inline classes is an Experimental feature. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.7.20 allows the underlying type of JVM inline classes to be a type parameter. The compiler maps it to Any? or, generally, to the upper bound of the type parameter.

Watch video online.

Consider the following example:

```kotlin
@JvmInline
value class UserId<T>(val value: T)

fun compute(s: UserId<String>) {} // Compiler generates fun compute-<hashcode>(s: Any?)
```

The function accepts the inline class as a parameter. The parameter is mapped to the upper bound, not the type argument.

To enable this feature, use the -language-version 1.8 compiler option.

We would appreciate your feedback on this feature in YouTrack.

More optimized cases of delegated properties

In Kotlin 1.6.0, we optimized the case of delegating to a property by omitting the $delegate field and generating immediate access to the referenced property. In 1.7.20, we've implemented this optimization for more cases. The $delegate field will now be omitted if a delegate is:

- A named object:

```kotlin
object NamedObject {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String = ...
}

val s: String by NamedObject
```

- A final val property with a backing field and a default getter in the same module:

```kotlin
val impl: ReadOnlyProperty<Any?, String> = ...

class A {
    val s: String by impl
}
```

- A constant expression, an enum entry, this, or null. Here's an example of this:

```kotlin
class A {
    operator fun getValue(thisRef: Any?, property: KProperty<*>) ...

    val s by this
}
```

Learn more about delegated properties.

We would appreciate your feedback on this feature in YouTrack.

Support for the JVM IR backend in kapt stub generating task

> Support for the JVM IR backend in the kapt stub generating task is an Experimental feature. It may be changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes.

Before 1.7.20, the kapt stub generating task used the old backend, and repeatable annotations didn't work with kapt. With Kotlin 1.7.20, we've added support for the JVM IR backend in the kapt stub generating task. This makes it possible to use all the newest Kotlin features with kapt, including repeatable annotations.

To use the IR backend in kapt, add the following option to your gradle.properties file:

```
kapt.use.jvm.ir=true
```

We would appreciate your feedback on this feature in YouTrack.

Kotlin/Native

Kotlin 1.7.20 comes with the new Kotlin/Native memory manager enabled by default and gives you the option to customize the Info.plist file:

- The new default memory manager
- Customizing the Info.plist file

The new Kotlin/Native memory manager enabled by default

This release brings further stability and performance improvements to the new memory manager, allowing us to promote the new memory manager to Beta.

The previous memory manager complicated writing concurrent and asynchronous code, including issues with implementing the kotlinx.coroutines library. This blocked the adoption of Kotlin Multiplatform Mobile because concurrency limitations created problems with sharing Kotlin code between iOS and Android platforms. The new memory manager finally paves the way to promote Kotlin Multiplatform Mobile to Beta.

The new memory manager also supports the compiler cache that makes compilation times comparable to previous releases. For more on the benefits of the new memory manager, see our original blog post for the preview version. You can find more technical details in the documentation.

Configuration and setup

Starting with Kotlin 1.7.20, the new memory manager is the default. Not much additional setup is required.

If you've already turned it on manually, you can remove the kotlin.native.binary.memoryModel=experimental option from your gradle.properties or binaryOptions["memoryModel"] = "experimental" from the build.gradle(.kts) file.

If necessary, you can switch back to the legacy memory manager with the kotlin.native.binary.memoryModel=strict option in your gradle.properties. However, compiler cache support is no longer available for the legacy memory manager, so compilation times might worsen.

Freezing

In the new memory manager, freezing is deprecated. Don't use it unless you need your code to work with the legacy manager (where freezing is still required). This may be helpful for library authors that need to maintain support for the legacy memory manager or developers who want to have a fallback if they encounter issues with the new memory manager.

In such cases, you can temporarily support code for both new and legacy memory managers. To ignore deprecation warnings, do one of the following:

- Annotate usages of the deprecated API with @OptIn(FreezingIsDeprecated::class).
- Apply languageSettings.optIn("kotlin.native.FreezingIsDeprecated") to all the Kotlin source sets in Gradle.
- Pass the compiler flag -opt-in=kotlin.native.FreezingIsDeprecated.

Calling Kotlin suspending functions from Swift/Objective-C

The new memory manager still restricts calling Kotlin suspend functions from Swift and Objective-C from threads other than the main one, but you can lift it with a

new Gradle option.

This restriction was originally introduced in the legacy memory manager due to cases where the code dispatched a continuation to be resumed on the original thread. If this thread didn't have a supported event loop, the task would never run, and the coroutine would never be resumed.

In certain cases, this restriction is no longer required, but a check of all the necessary conditions can't be easily implemented. Because of this, we decided to keep it in the new memory manager while introducing an option for you to disable it. For this, add the following option to your gradle.properties:

```
kotlin.native.binary.objcExportSuspendFunctionLaunchThreadRestriction=none
```

> Do not add this option if you use the native-mt version of kotlinx.coroutines or other libraries that have the same "dispatch to the original thread" approach.

The Kotlin team is very grateful to Ahmed El-Helw for implementing this option.

Leave your feedback

This is a significant change to our ecosystem. We would appreciate your feedback to help make it even better.

Try the new memory manager on your projects and share feedback in our issue tracker, YouTrack.

Customizing the Info.plist file

When producing a framework, the Kotlin/Native compiler generates the information property list file, Info.plist. Previously, it was cumbersome to customize its contents. With Kotlin 1.7.20, you can directly set the following properties:

Property	Binary option
CFBundleIdentifier	bundleId
CFBundleShortVersionString	bundleShortVersionString
CFBundleVersion	bundleVersion

To do that, use the corresponding binary option. Pass the -Xbinary=$option=$value compiler flag or set the binaryOption(option, value) Gradle DSL for the necessary framework.

The Kotlin team is very grateful to Mads Ager for implementing this feature.

Kotlin/JS

Kotlin/JS has received some enhancements that improve the developer experience and boost performance:

- Klib generation is faster in both incremental and clean builds, thanks to efficiency improvements for the loading of dependencies.
- Incremental compilation for development binaries has been reworked, resulting in major improvements in clean build scenarios, faster incremental builds, and stability fixes.
- We've improved .d.ts generation for nested objects, sealed classes, and optional parameters in constructors.

Gradle

The updates for the Kotlin Gradle plugin are focused on compatibility with the new Gradle features and the latest Gradle versions.

Kotlin 1.7.20 contains changes to support Gradle 7.1. Deprecated methods and properties were removed or replaced, reducing the number of deprecation warnings produced by the Kotlin Gradle plugin and unblocking future support for Gradle 8.0.

There are, however, some potentially breaking changes that may need your attention:

Target configuration

- org.jetbrains.kotlin.gradle.dsl.SingleTargetExtension now has a generic parameter, SingleTargetExtension<T : KotlinTarget>.
- The kotlin.targets.fromPreset() convention has been deprecated. Instead, you can still use kotlin.targets { fromPreset() }, but we recommend using more specialized ways to create targets.
- Target accessors auto-generated by Gradle are no longer available inside the kotlin.targets { } block. Please use the findByName("targetName") method instead.

  Note that such accessors are still available in the case of kotlin.targets, for example, kotlin.targets.linuxX64.

Source directories configuration

The Kotlin Gradle plugin now adds Kotlin SourceDirectorySet as a kotlin extension to Java's SourceSet group. This makes it possible to configure source directories in the build.gradle.kts file similarly to how they are configured in Java, Groovy, and Scala:

```
sourceSets {
    main {
        kotlin {
            java.setSrcDirs(listOf("src/java"))
            kotlin.setSrcDirs(listOf("src/kotlin"))
        }
    }
}
```

You no longer need to use a deprecated Gradle convention and specify the source directories for Kotlin.

Remember that you can also use the kotlin extension to access KotlinSourceSet:

```
kotlin {
    sourceSets {
        main {
        // ...
        }
    }
}
```

New method for JVM toolchain configuration

This release provides a new jvmToolchain() method for enabling the JVM toolchain feature. If you don't need any additional configuration fields, such as implementation or vendor, you can use this method from the Kotlin extension:

```
kotlin {
    jvmToolchain(17)
}
```

This simplifies the Kotlin project setup process without any additional configuration. Before this release, you could specify the JDK version only in the following way:

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(17))
    }
}
```

Standard library

Kotlin 1.7.20 offers new extension functions for the java.nio.file.Path class, which allows you to walk through a file tree:

- walk() lazily traverses the file tree rooted at the specified path.
- fileVisitor() makes it possible to create a FileVisitor separately. FileVisitor defines actions on directories and files when traversing them.
- visitFileTree(fileVisitor: FileVisitor, ...) consumes a ready FileVisitor and uses java.nio.file.Files.walkFileTree() under the hood.

* visitFileTree(..., builderAction: FileVisitorBuilder.() -> Unit) creates a FileVisitor with the builderAction and calls the visitFileTree(fileVisitor, ...) function.
* FileVisitResult, return type of FileVisitor, has the CONTINUE default value that continues the processing of the file.

The new extension functions for java.nio.file.Path are Experimental. They may be changed at any time. Opt-in is required (see details below), and you should use them only for evaluation purposes.

Here are some things you can do with these new extension functions:

* Explicitly create a FileVisitor and then use:

```
val cleanVisitor = fileVisitor {
    onPreVisitDirectory { directory, attributes ->
        // Some logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Some logic on visiting files
        FileVisitResult.CONTINUE
    }
}

// Some logic may go here

projectDirectory.visitFileTree(cleanVisitor)
```

* Create a FileVisitor with the builderAction and use it immediately:

```
projectDirectory.visitFileTree {
// Definition of the builderAction:
    onPreVisitDirectory { directory, attributes ->
        // Some logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Some logic on visiting files
        FileVisitResult.CONTINUE
    }
}
```

* Traverse a file tree rooted at the specified path with the walk() function:

```
@OptIn(kotlin.io.path.ExperimentalPathApi::class)
fun traverseFileTree() {
    val cleanVisitor = fileVisitor {
        onPreVisitDirectory { directory, _ ->
            if (directory.name == "build") {
                directory.toFile().deleteRecursively()
                FileVisitResult.SKIP_SUBTREE
            } else {
                FileVisitResult.CONTINUE
            }
        }

        onVisitFile { file, _ ->
            if (file.extension == "class") {
                file.deleteExisting()
            }
            FileVisitResult.CONTINUE
        }
    }

    val rootDirectory = createTempDirectory("Project")

    rootDirectory.resolve("src").let { srcDirectory ->
        srcDirectory.createDirectory()
        srcDirectory.resolve("A.kt").createFile()
        srcDirectory.resolve("A.class").createFile()
    }

    rootDirectory.resolve("build").let { buildDirectory ->
```

```
        buildDirectory.createDirectory()
        buildDirectory.resolve("Project.jar").createFile()
    }


// Use walk function:
    val directoryStructure = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    assertPrints(directoryStructure, "[, build, build/Project.jar, src, src/A.class, src/A.kt]")

    rootDirectory.visitFileTree(cleanVisitor)

    val directoryStructureAfterClean = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    assertPrints(directoryStructureAfterClean, "[, src, src/A.kt]")
//sampleEnd
}
```

As is usual for an experimental API, the new extensions require an opt-in: @OptIn(kotlin.io.path.ExperimentalPathApi::class) or @kotlin.io.path.ExperimentalPathApi. Alternatively, you can use a compiler option: -opt-in=kotlin.io.path.ExperimentalPathApi.

We would appreciate your feedback on the walk() function and the visit extension functions in YouTrack.

Documentation updates

Since the previous release, the Kotlin documentation has received some notable changes:

Revamped and improved pages

- Basic types overview – learn about the basic types used in Kotlin: numbers, Booleans, characters, strings, arrays, and unsigned integer numbers.
- IDEs for Kotlin development – see the list of IDEs with official Kotlin support and tools that have community-supported plugins.

New articles in the Kotlin Multiplatform journal

- Native and cross-platform app development: how to choose? – check out our overview and advantages of cross-platform app development and the native approach.
- The six best cross-platform app development frameworks – read about the key aspects to help you choose the right framework for your cross-platform project.

New and updated tutorials

- Get started with Kotlin Multiplatform Mobile – learn about cross-platform mobile development with Kotlin and create an app that works on both Android and iOS.
- Build a full-stack web app with Kotlin Multiplatform – create an app using Kotlin throughout the whole stack, with a Kotlin/JVM server part and a Kotlin/JS web client.
- Build a web application with React and Kotlin/JS – create a browser app exploring Kotlin's DSLs and features of a typical React program.

Changes in release documentation

We no longer provide a list of recommended kotlinx libraries for each release. This list included only the versions recommended and tested with Kotlin itself. It didn't take into account that some libraries depend on each other and require a special kotlinx version, which may differ from the recommended Kotlin version.

We're working on finding a way to provide information on how libraries interrelate and depend on each other so that it will be clear which kotlinx library version you should use when you upgrade the Kotlin version in your project.

Install Kotlin 1.7.20

IntelliJ IDEA 2021.3, 2022.1, and 2022.2 automatically suggest updating the Kotlin plugin to 1.7.20.

For Android Studio Dolphin (213), Electric Eel (221), and Flamingo (222), the Kotlin plugin 1.7.20 will be delivered with upcoming Android Studios updates.

The new command-line compiler is available for download on the GitHub release page.

Compatibility guide for Kotlin 1.7.20

Although Kotlin 1.7.20 is an incremental release, there are still incompatible changes we had to make to limit spread of the issues introduced in Kotlin 1.7.0.

Find the detailed list of such changes in the Compatibility guide for Kotlin 1.7.20.

What's new in Kotlin 1.7.0

Released: 9 June 2022

Kotlin 1.7.0 has been released. It unveils the Alpha version of the new Kotlin/JVM K2 compiler, stabilizes language features, and brings performance improvements for the JVM, JS, and Native platforms.

Here is a list of the major updates in this version:

- The new Kotlin K2 compiler is in Alpha now, and it offers serious performance improvements. It is available only for the JVM, and none of the compiler plugins, including kapt, work with it.
- A new approach to the incremental compilation in Gradle. Incremental compilation is now also supported for changes made inside dependent non-Kotlin modules and is compatible with Gradle.
- We've stabilized opt-in requirement annotations, definitely non-nullable types, and builder inference.
- There's now an underscore operator for type args. You can use it to automatically infer a type of argument when other types are specified.
- This release allows implementation by delegation to an inlined value of an inline class. You can now create lightweight wrappers that do not allocate memory in most cases.

You can also find a short overview of the changes in this video:

Watch video online.

New Kotlin K2 compiler for the JVM in Alpha

This Kotlin release introduces the Alpha version of the new Kotlin K2 compiler. The new compiler aims to speed up the development of new language features, unify all of the platforms Kotlin supports, bring performance improvements, and provide an API for compiler extensions.

We've already published some detailed explanations of our new compiler and its benefits:

- The Road to the New Kotlin Compiler

* K2 Compiler: a Top-Down View

It's important to point out that with the Alpha version of the new K2 compiler we were primarily focused on performance improvements, and it only works with JVM projects. It doesn't support Kotlin/JS, Kotlin/Native, or other multi-platform projects, and none of compiler plugins, including kapt, work with it.

Our benchmarks show some outstanding results on our internal projects:

Project	Current Kotlin compiler performance	New K2 Kotlin compiler performance	Performance boost
Kotlin	2.2 KLOC/s	4.8 KLOC/s	~ x2.2
YouTrack	1.8 KLOC/s	4.2 KLOC/s	~ x2.3
IntelliJ IDEA	1.8 KLOC/s	3.9 KLOC/s	~ x2.2
Space	1.2 KLOC/s	2.8 KLOC/s	~ x2.3

> The KLOC/s performance numbers stand for the number of thousands of lines of code that the compiler processes per second.

You can check out the performance boost on your JVM projects and compare it with the results of the old compiler. To enable the Kotlin K2 compiler, use the following compiler option:

```
-Xuse-k2
```

Also, the K2 compiler includes a number of bugfixes. Please note that even issues with State: Open from this list are in fact fixed in K2.

The next Kotlin releases will improve the stability of the K2 compiler and provide more features, so stay tuned!

If you face any performance issues with the Kotlin K2 compiler, please report them to our issue tracker.

Language

Kotlin 1.7.0 introduces support for implementation by delegation and a new underscore operator for type arguments. It also stabilizes several language features introduced as previews in previous releases:

* Implementation by delegation to inlined value of inline class
* Underscore operator for type arguments
* Stable builder inference
* Stable opt-in requirements
* Stable definitely non-nullable types

Allow implementation by delegation to an inlined value of an inline class

If you want to create a lightweight wrapper for a value or class instance, it's necessary to implement all interface methods by hand. Implementation by delegation solves this issue, but it did not work with inline classes before 1.7.0. This restriction has been removed, so you can now create lightweight wrappers that do not allocate memory in most cases.

```
interface Bar {
    fun foo() = "foo"
}

@JvmInline
value class BarWrapper(val bar: Bar): Bar by bar
```

```kotlin
fun main() {
    val bw = BarWrapper(object: Bar {})
    println(bw.foo())
}
```

Underscore operator for type arguments

Kotlin 1.7.0 introduces an underscore operator, _, for type arguments. You can use it to automatically infer a type argument when other types are specified:

```kotlin
abstract class SomeClass<T> {
    abstract fun execute(): T
}

class SomeImplementation : SomeClass<String>() {
    override fun execute(): String = "Test"
}

class OtherImplementation : SomeClass<Int>() {
    override fun execute(): Int = 42
}

object Runner {
    inline fun <reified S: SomeClass<T>, T> run(): T {
        return S::class.java.getDeclaredConstructor().newInstance().execute()
    }
}

fun main() {
    // T is inferred as String because SomeImplementation derives from SomeClass<String>
    val s = Runner.run<SomeImplementation, _>()
    assert(s == "Test")

    // T is inferred as Int because OtherImplementation derives from SomeClass<Int>
    val n = Runner.run<OtherImplementation, _>()
    assert(n == 42)
}
```

> You can use the underscore operator in any position in the variables list to infer a type argument.

Stable builder inference

Builder inference is a special kind of type inference that is useful when calling generic builder functions. It helps the compiler infer the type arguments of a call using the type information about other calls inside its lambda argument.

Starting with 1.7.0, builder inference is automatically activated if a regular type inference cannot get enough information about a type without specifying the -Xenable-builder-inference compiler option, which was introduced in 1.6.0.

Learn how to write custom generic builders.

Stable opt-in requirements

Opt-in requirements are now Stable and do not require additional compiler configuration.

Before 1.7.0, the opt-in feature itself required the argument -opt-in=kotlin.RequiresOptIn to avoid a warning. It no longer requires this; however, you can still use the compiler argument -opt-in to opt-in for other annotations, module-wide.

Stable definitely non-nullable types

In Kotlin 1.7.0, definitely non-nullable types have been promoted to Stable. They provide better interoperability when extending generic Java classes and interfaces.

You can mark a generic type parameter as definitely non-nullable at the use site with the new syntax T & Any. The syntactic form comes from the notation for intersection types and is now limited to a type parameter with nullable upper bounds on the left side of & and a non-nullable Any on the right side:

```kotlin
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y

fun main() {
    // OK
    elvisLike<String>("", "").length
```

```
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String>("", null).length

    // OK
    elvisLike<String?>(null, "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String?>(null, null).length
}
```

Learn more about definitely non-nullable types in this KEEP.

Kotlin/JVM

This release brings performance improvements for the Kotlin/JVM compiler and a new compiler option. Additionally, callable references to functional interface constructors have become Stable. Note that since 1.7.0, the default target version for Kotlin/JVM compilations is 1.8.

- Compiler performance optimizations
- New compiler option -Xjdk-release
- Stable callable references to functional interface constructors
- Removed the JVM target version 1.6

Compiler performance optimizations

Kotlin 1.7.0 introduces performance improvements for the Kotlin/JVM compiler. According to our benchmarks, compilation time has been reduced by 10% on average compared to Kotlin 1.6.0. Projects with lots of usages of inline functions, for example, projects using kotlinx.html, will compile faster thanks to the improvements to the bytecode postprocessing.

New compiler option: -Xjdk-release

Kotlin 1.7.0 presents a new compiler option, -Xjdk-release. This option is similar to the javac's command-line --release option. The -Xjdk-release option controls the target bytecode version and limits the API of the JDK in the classpath to the specified Java version. For example, kotlinc -Xjdk-release=1.8 won't allow referencing java.lang.Module even if the JDK in the dependencies is version 9 or higher.

> This option is not guaranteed to be effective for each JDK distribution.

Please leave your feedback on this YouTrack ticket.

Stable callable references to functional interface constructors

Callable references to functional interface constructors are now Stable. Learn how to migrate from an interface with a constructor function to a functional interface using callable references.

Please report any issues you find in YouTrack.

Removed JVM target version 1.6

The default target version for Kotlin/JVM compilations is 1.8. The 1.6 target has been removed.

Please migrate to JVM target 1.8 or above. Learn how to update the JVM target version for:

- Gradle
- Maven
- The command-line compiler

Kotlin/Native

Kotlin 1.7.0 includes changes to Objective-C and Swift interoperability and stabilizes features that were introduced in previous releases. It also brings performance

improvements for the new memory manager along with other updates:

- Performance improvements for the new memory manager
- Unified compiler plugin ABI with JVM and JS IR backends
- Support for standalone Android executables
- Interop with Swift async/await: returning Void instead of KotlinUnit
- Prohibited undeclared exceptions through Objective-C bridges
- Improved CocoaPods integration
- Overriding of the Kotlin/Native compiler download URL

Performance improvements for the new memory manager

> The new Kotlin/Native memory manager is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback in YouTrack.

The new memory manager is still in Alpha, but it is on its way to becoming Stable. This release delivers significant performance improvements for the new memory manager, especially in garbage collection (GC). In particular, concurrent implementation of the sweep phase, introduced in 1.6.20, is now enabled by default. This helps reduce the time the application is paused for GC. The new GC scheduler is better at choosing the GC frequency, especially for larger heaps.

Also, we've specifically optimized debug binaries, ensuring that the proper optimization level and link-time optimizations are used in the implementation code of the memory manager. This helped us improve execution time by roughly 30% for debug binaries on our benchmarks.

Try using the new memory manager in your projects to see how it works, and share your feedback with us in YouTrack.

Unified compiler plugin ABI with JVM and JS IR backends

Starting with Kotlin 1.7.0, the Kotlin Multiplatform Gradle plugin uses the embeddable compiler jar for Kotlin/Native by default. This feature was announced in 1.6.0 as Experimental, and now it's Stable and ready to use.

This improvement is very handy for library authors, as it improves the compiler plugin development experience. Before this release, you had to provide separate artifacts for Kotlin/Native, but now you can use the same compiler plugin artifacts for Native and other supported platforms.

> This feature might require plugin developers to take migration steps for their existing plugins.
>
> Learn how to prepare your plugin for the update in this YouTrack issue.

Support for standalone Android executables

Kotlin 1.7.0 provides full support for generating standard executables for Android Native targets. It was introduced in 1.6.20, and now it's enabled by default.

If you want to roll back to the previous behavior when Kotlin/Native generated shared libraries, use the following setting:

```
binaryOptions["androidProgramType"] = "nativeActivity"
```

Interop with Swift async/await: returning Void instead of KotlinUnit

Kotlin suspend functions now return the Void type instead of KotlinUnit in Swift. This is the result of the improved interop with Swift's async/await. This feature was introduced in 1.6.20, and this release enables this behavior by default.

You don't need to use the kotlin.native.binary.unitSuspendFunctionObjCExport=proper property anymore to return the proper type for such functions.

Prohibited undeclared exceptions through Objective-C bridges

When you call Kotlin code from Swift/Objective-C code (or vice versa) and this code throws an exception, it should be handled by the code where the exception occurred, unless you specifically allowed the forwarding of exceptions between languages with proper conversion (for example, using the @Throws annotation).

Previously, Kotlin had another unintended behavior where undeclared exceptions could "leak" from one language to another in some cases. Kotlin 1.7.0 fixes that issue, and now such cases lead to program termination.

So, for example, if you have a { throw Exception() } lambda in Kotlin and call it from Swift, in Kotlin 1.7.0 it will terminate as soon as the exception reaches the Swift code. In previous Kotlin versions, such an exception could leak to the Swift code.

The @Throws annotation continues to work as before.

Improved CocoaPods integration

Starting with Kotlin 1.7.0, you no longer need to install the cocoapods-generate plugin if you want to integrate CocoaPods in your projects.

Previously, you needed to install both the CocoaPods dependency manager and the cocoapods-generate plugin to use CocoaPods, for example, to handle iOS dependencies in Kotlin Multiplatform Mobile projects.

Now setting up the CocoaPods integration is easier, and we've resolved the issue when cocoapods-generate couldn't be installed on Ruby 3 and later. Now the newest Ruby versions that work better on Apple M1 are also supported.

See how to set up the initial CocoaPods integration.

Overriding the Kotlin/Native compiler download URL

Starting with Kotlin 1.7.0, you can customize the download URL for the Kotlin/Native compiler. This is useful when external links on the CI are forbidden.

To override the default base URL https://download.jetbrains.com/kotlin/native/builds, use the following Gradle property:

```
kotlin.native.distribution.baseDownloadUrl=https://example.com
```

> The downloader will append the native version and target OS to this base URL to ensure it downloads the actual compiler distribution.

Kotlin/JS

Kotlin/JS is receiving further improvements to the JS IR compiler backend along with other updates that can make your development experience better:

- Performance improvements for the new IR backend
- Minification for member names when using IR
- Support for older browsers via polyfills in the IR backend
- Dynamically load JavaScript modules from js expressions
- Specify environment variables for JavaScript test runners

Performance improvements for the new IR backend

This release has some major updates that should improve your development experience:

- Incremental compilation performance of Kotlin/JS has been significantly improved. It takes less time to build your JS projects. Incremental rebuilds should now be roughly on par with the legacy backend in many cases now.
- The Kotlin/JS final bundle requires less space, as we have significantly reduced the size of the final artifacts. We've measured up to a 20% reduction in the production bundle size compared to the legacy backend for some large projects.
- Type checking for interfaces has been improved by orders of magnitude.
- Kotlin generates higher-quality JS code

Minification for member names when using IR

The Kotlin/JS IR compiler now uses its internal information about the relationships of your Kotlin classes and functions to apply more efficient minification, shortening the names of functions, properties, and classes. This shrinks the resulting bundled applications.

This type of minification is automatically applied when you build your Kotlin/JS application in production mode and is enabled by default. To disable member name

minification, use the -Xir-minimized-member-names compiler flag:

```
kotlin {
    js(IR) {
        compilations.all {
            compileKotlinTask.kotlinOptions.freeCompilerArgs += listOf("-Xir-minimized-member-names=false")
        }
    }
}
```

Support for older browsers via polyfills in the IR backend

The IR compiler backend for Kotlin/JS now includes the same polyfills as the legacy backend. This allows code compiled with the new compiler to run in older browsers that do not support all the methods from ES2015 used by the Kotlin standard library. Only those polyfills actually used by the project are included in the final bundle, which minimizes their potential impact on the bundle size.

This feature is enabled by default when using the IR compiler, and you don't need to configure it.

Dynamically load JavaScript modules from js expressions

When working with the JavaScript modules, most applications use static imports, whose use is covered with the JavaScript module integration. However, Kotlin/JS was missing a mechanism to load JavaScript modules dynamically at runtime in your applications.

Starting with Kotlin 1.7.0, the import statement from JavaScript is supported in js blocks, allowing you to dynamically bring packages into your application at runtime:

```
val myPackage = js("import('my-package')")
```

Specify environment variables for JavaScript test runners

To tune Node.js package resolution or pass external information to Node.js tests, you can now specify environment variables used by the JavaScript test runners. To define an environment variable, use the environment() function with a key-value pair inside the testTask block in your build script:

```
kotlin {
    js {
        nodejs {
            testTask {
                environment("key", "value")
            }
        }
    }
}
```

Standard library

In Kotlin 1.7.0, the standard library has received a range of changes and improvements. They introduce new features, stabilize experimental ones, and unify support for named capturing groups for Native, JS, and the JVM:

- min() and max() collection functions return as non-nullable
- Regular expression matching at specific indices
- Extended support of previous language and API versions
- Access to annotations via reflection
- Stable deep recursive functions
- Time marks based on inline classes for default time source
- New experimental extension functions for Java Optionals
- Support for named capturing groups in JS and Native

min() and max() collection functions return as non-nullable

In Kotlin 1.4.0, we renamed the min() and max() collection functions to minOrNull() and maxOrNull(). These new names better reflect their behavior – returning null if the receiver collection is empty. It also helped align the functions' behavior with naming conventions used throughout the Kotlin collections API.

The same was true of minBy(), maxBy(), minWith(), and maxWith(), which all got their *OrNull() synonyms in Kotlin 1.4.0. Older functions affected by this change were gradually deprecated.

Kotlin 1.7.0 reintroduces the original function names, but with a non-nullable return type. The new min(), max(), minBy(), maxBy(), minWith(), and maxWith() functions now strictly return the collection element or throw an exception.

```
fun main() {
    val numbers = listOf<Int>()
    println(numbers.maxOrNull()) // "null"
    println(numbers.max()) // "Exception in... Collection is empty."
}
```

Regular expression matching at specific indices

The Regex.matchAt() and Regex.matchesAt() functions, introduced in 1.5.30, are now Stable. They provide a way to check whether a regular expression has an exact match at a particular position in a String or CharSequence.

matchesAt() checks for a match and returns a boolean result:

```
fun main() {
    val releaseText = "Kotlin 1.7.0 is on its way!"
    // regular expression: one digit, dot, one digit, dot, one or more digits
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()

    println(versionRegex.matchesAt(releaseText, 0)) // "false"
    println(versionRegex.matchesAt(releaseText, 7)) // "true"
}
```

matchAt() returns the match if it's found, or null if it isn't:

```
fun main() {
    val releaseText = "Kotlin 1.7.0 is on its way!"
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()

    println(versionRegex.matchAt(releaseText, 0)) // "null"
    println(versionRegex.matchAt(releaseText, 7)?.value) // "1.7.0"
}
```

We'd be grateful for your feedback on this YouTrack issue.

Extended support for previous language and API versions

To support library authors developing libraries that are meant to be consumable in a wide range of previous Kotlin versions, and to address the increased frequency of major Kotlin releases, we have extended our support for previous language and API versions.

With Kotlin 1.7.0, we're supporting three previous language and API versions rather than two. This means Kotlin 1.7.0 supports the development of libraries targeting Kotlin versions down to 1.4.0. For more information on backward compatibility, see Compatibility modes.

Access to annotations via reflection

The KAnnotatedElement.findAnnotations() extension function, which was first introduced in 1.6.0, is now Stable. This reflection function returns all annotations of a given type on an element, including individually applied and repeated annotations.

```
@Repeatable
annotation class Tag(val name: String)

@Tag("First Tag")
@Tag("Second Tag")
fun taggedFunction() {
    println("I'm a tagged function!")
}

fun main() {
    val x = ::taggedFunction
```

```
    val foo = x as KAnnotatedElement
    println(foo.findAnnotations<Tag>()) // [@Tag(name=First Tag), @Tag(name=Second Tag)]
}
```

Stable deep recursive functions

Deep recursive functions have been available as an experimental feature since Kotlin 1.4.0, and they are now Stable in Kotlin 1.7.0. Using DeepRecursiveFunction, you can define a function that keeps its stack on the heap instead of using the actual call stack. This allows you to run very deep recursive computations. To call a deep recursive function, invoke it.

In this example, a deep recursive function is used to calculate the depth of a binary tree recursively. Even though this sample function calls itself recursively 100,000 times, no StackOverflowError is thrown:

```
class Tree(val left: Tree?, val right: Tree?)

val calculateDepth = DeepRecursiveFunction<Tree?, Int> { t ->
    if (t == null) 0 else maxOf(
        callRecursive(t.left),
        callRecursive(t.right)
    ) + 1
}

fun main() {
    // Generate a tree with a depth of 100_000
    val deepTree = generateSequence(Tree(null, null)) { prev ->
        Tree(prev, null)
    }.take(100_000).last()

    println(calculateDepth(deepTree)) // 100000
}
```

Consider using deep recursive functions in your code where your recursion depth exceeds 1000 calls.

Time marks based on inline classes for default time source

Kotlin 1.7.0 improves the performance of time measurement functionality by changing the time marks returned by TimeSource.Monotonic into inline value classes. This means that calling functions like markNow(), elapsedNow(), measureTime(), and measureTimedValue() doesn't allocate wrapper classes for their TimeMark instances. Especially when measuring a piece of code that is part of a hot path, this can help minimize the performance impact of the measurement:

```
@OptIn(ExperimentalTime::class)
fun main() {
    val mark = TimeSource.Monotonic.markNow() // Returned `TimeMark` is inline class
    val elapsedDuration = mark.elapsedNow()
}
```

> This optimization is only available if the time source from which the TimeMark is obtained is statically known to be TimeSource.Monotonic.

New experimental extension functions for Java Optionals

Kotlin 1.7.0 comes with new convenience functions that simplify working with Optional classes in Java. These new functions can be used to unwrap and convert optional objects on the JVM and help make working with Java APIs more concise.

The getOrNull(), getOrDefault(), and getOrElse() extension functions allow you to get the value of an Optional if it's present. Otherwise, you get null, a default value, or a value returned by a function, respectively:

```
val presentOptional = Optional.of("I'm here!")

println(presentOptional.getOrNull())
// "I'm here!"

val absentOptional = Optional.empty<String>()

println(absentOptional.getOrNull())
// null
println(absentOptional.getOrDefault("Nobody here!"))
// "Nobody here!"
println(absentOptional.getOrElse {
    println("Optional was absent!")
```

```
    "Default value!"
})
// "Optional was absent!"
// "Default value!"
```

The toList(), toSet(), and asSequence() extension functions convert the value of a present Optional to a list, set, or sequence, or return an empty collection otherwise. The toCollection() extension function appends the Optional value to an already existing destination collection:

```
val presentOptional = Optional.of("I'm here!")
val absentOptional = Optional.empty<String>()
println(presentOptional.toList() + "," + absentOptional.toList())
// ["I'm here!"], []
println(presentOptional.toSet() + "," + absentOptional.toSet())
// ["I'm here!"], []
val myCollection = mutableListOf<String>()
absentOptional.toCollection(myCollection)
println(myCollection)
// []
presentOptional.toCollection(myCollection)
println(myCollection)
// ["I'm here!"]
val list = listOf(presentOptional, absentOptional).flatMap { it.asSequence() }
println(list)
// ["I'm here!"]
```

These extension functions are being introduced as Experimental in Kotlin 1.7.0. You can learn more about Optional extensions in this KEEP. As always, we welcome your feedback in the Kotlin issue tracker.

Support for named capturing groups in JS and Native

Starting with Kotlin 1.7.0, named capturing groups are supported not only on the JVM, but on the JS and Native platforms as well.

To give a name to a capturing group, use the (?<name>group) syntax in your regular expression. To get the text matched by a group, call the newly introduced MatchGroupCollection.get() function and pass the group name.

Retrieve matched group value by name

Consider this example for matching city coordinates. To get a collection of groups matched by the regular expression, use groups. Compare retrieving a group's contents by its number (index) and by its name using value:

```
fun main() {
    val regex = "\\b(?<city>[A-Za-z\\s]+),\\s(?<state>[A-Z]{2}):\\s(?<areaCode>[0-9]{3})\\b".toRegex()
    val input = "Coordinates: Austin, TX: 123"
    val match = regex.find(input)!!
    println(match.groups["city"]?.value) // "Austin" — by name
    println(match.groups[2]?.value) // "TX" — by number
}
```

Named backreferencing

You can now also use group names when backreferencing groups. Backreferences match the same text that was previously matched by a capturing group. For this, use the \k<name> syntax in your regular expression:

```
fun backRef() {
    val regex = "(?<title>\\w+), yes \\k<title>".toRegex()
    val match = regex.find("Do you copy? Sir, yes Sir!")!!
    println(match.value) // "Sir, yes Sir"
    println(match.groups["title"]?.value) // "Sir"
}
```

Named groups in replacement expressions

Named group references can be used with replacement expressions. Consider the replace() function that substitutes all occurrences of the specified regular expression in the input with a replacement expression, and the replaceFirst() function that swaps the first match only.

Occurrences of ${name} in the replacement string are substituted with the subsequences corresponding to the captured groups with the specified name. You can compare replacements in group references by name and index:

```kotlin
fun dateReplace() {
    val dateRegex = Regex("(?<dd>\\d{2})-(?<mm>\\d{2})-(?<yyyy>\\d{4})")
    val input = "Date of birth: 27-04-2022"
    println(dateRegex.replace(input, "\${yyyy}-\${mm}-\${dd}")) // "Date of birth: 2022-04-27" — by name
    println(dateRegex.replace(input, "\$3-\$2-\$1")) // "Date of birth: 2022-04-27" — by number
}
```

Gradle

This release introduces new build reports, support for Gradle plugin variants, new statistics in kapt, and a lot more:

- A new approach to incremental compilation
- New build reports for tracking compiler performance
- Changes to the minimum supported versions of Gradle and the Android Gradle plugin
- Support for Gradle plugin variants
- Updates in the Kotlin Gradle plugin API
- Availability of the sam-with-receiver plugin via the plugins API
- Changes in compile tasks
- New statistics of generated files by each annotation processor in kapt
- Deprecation of the kotlin.compiler.execution.strategy system property
- Removal of deprecated options, methods, and plugins

A new approach to incremental compilation

> The new approach to incremental compilation is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below). We encourage you to use it only for evaluation purposes, and we would appreciate your feedback in YouTrack.

In Kotlin 1.7.0, we've reworked incremental compilation for cross-module changes. Now incremental compilation is also supported for changes made inside dependent non-Kotlin modules, and it is compatible with the Gradle build cache. Support for compilation avoidance has also been improved.

We expect you'll see the most significant benefit of the new approach if you use the build cache or frequently make changes in non-Kotlin Gradle modules. Our tests for the Kotlin project on the kotlin-gradle-plugin module show an improvement of greater than 80% for the changes after the cache hit.

To try this new approach, set the following option in your gradle.properties:

```
kotlin.incremental.useClasspathSnapshot=true
```

> The new approach to incremental compilation is currently available for the JVM backend in the Gradle build system only.

Learn how the new approach to incremental compilation is implemented under the hood in this blog post.

Our plan is to stabilize this technology and add support for other backends (JS, for instance) and build systems. We'd appreciate your reports in YouTrack about any issues or strange behavior you encounter in this compilation scheme. Thank you!

The Kotlin team is very grateful to Ivan Gavrilovic, Hung Nguyen, Cédric Champeau, and other external contributors for their help.

Build reports for Kotlin compiler tasks

> Kotlin build reports are Experimental. They may be dropped or changed at any time. Opt-in is required (see details below). Use them only for evaluation purposes. We appreciate your feedback on them in YouTrack.

Kotlin 1.7.0 introduces build reports that help track compiler performance. Reports contain the durations of different compilation phases and reasons why compilation couldn't be incremental.

Build reports come in handy when you want to investigate issues with compiler tasks, for example:

- When the Gradle build takes too much time and you want to understand the root cause of the poor performance.
- When the compilation time for the same project differs, sometimes taking seconds, sometimes taking minutes.

To enable build reports, declare where to save the build report output in gradle.properties:

```
kotlin.build.report.output=file
```

The following values (and their combinations) are available:

- file saves build reports in a local file.
- build_scan saves build reports in the custom values section of the build scan.

  > The Gradle Enterprise plugin limits the number of custom values and their length. In big projects, some values could be lost.

- http posts build reports using HTTP(S). The POST method sends metrics in the JSON format. Data may change from version to version. You can see the current version of the sent data in the Kotlin repository.

There are two common cases that analyzing build reports for long-running compilations can help you resolve:

- The build wasn't incremental. Analyze the reasons and fix underlying problems.
- The build was incremental, but took too much time. Try to reorganize source files — split big files, save separate classes in different files, refactor large classes, declare top-level functions in different files, and so on.

Learn more about new build reports in this blog post.

You are welcome to try using build reports in your infrastructure. If you have any feedback, encounter any issues, or want to suggest improvements, please don't hesitate to report them in our issue tracker. Thank you!

Bumping minimum supported versions

Starting with Kotlin 1.7.0, the minimum supported Gradle version is 6.7.1. We had to raise the version to support Gradle plugin variants and the new Gradle API. In the future, we should not have to raise the minimum supported version as often, thanks to the Gradle plugin variants feature.

Also, the minimal supported Android Gradle plugin version is now 3.6.4.

Support for Gradle plugin variants

Gradle 7.0 introduced a new feature for Gradle plugin authors — plugins with variants. This feature makes it easier to add support for new Gradle features while maintaining compatibility for Gradle versions below 7.1. Learn more about variant selection in Gradle.

With Gradle plugin variants, we can ship different Kotlin Gradle plugin variants for different Gradle versions. The goal is to support the base Kotlin compilation in the main variant, which corresponds to the oldest supported versions of Gradle. Each variant will have implementations for Gradle features from a corresponding release. The latest variant will support the widest Gradle feature set. With this approach, we can extend support for older Gradle versions with limited functionality.

Currently, there are only two variants of the Kotlin Gradle plugin:

- main for Gradle versions 6.7.1–6.9.3
- gradle70 for Gradle versions 7.0 and higher

In future Kotlin releases, we may add more.

To check which variant your build uses, enable the --info log level and find a string in the output starting with Using Kotlin Gradle plugin, for example, Using Kotlin

Gradle plugin main variant.

Here are workarounds for some known issues with variant selection in Gradle:

- ResolutionStrategy in pluginManagement is not working for plugins with multivariants
- Plugin variants are ignored when a plugin is added as the buildSrc common dependency

Leave your feedback on this YouTrack ticket.

Updates in the Kotlin Gradle plugin API

The Kotlin Gradle plugin API artifact has received several improvements:

- There are new interfaces for Kotlin/JVM and Kotlin/kapt tasks with user-configurable inputs.
- There is a new KotlinBasePlugin interface that all Kotlin plugins inherit from. Use this interface when you want to trigger some configuration action whenever any Kotlin Gradle plugin (JVM, JS, Multiplatform, Native, and other platforms) is applied:

  ```
  project.plugins.withType<org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin>() {
      // Configure your action here
  }
  ```

  You can leave your feedback about the KotlinBasePlugin in this YouTrack ticket.
- We've laid the groundwork for the Android Gradle plugin to configure Kotlin compilation within itself, meaning you won't need to add the Kotlin Android Gradle plugin to your build. Follow Android Gradle Plugin release announcements to learn about the added support and try it out!

The sam-with-receiver plugin is available via the plugins API

The sam-with-receiver compiler plugin is now available via the Gradle plugins DSL:

```
plugins {
    id("org.jetbrains.kotlin.plugin.sam.with.receiver") version "$kotlin_version"
}
```

Changes in compile tasks

Compile tasks have received lots of changes in this release:

- Kotlin compile tasks no longer inherit the Gradle AbstractCompile task. They inherit only the DefaultTask.
- The AbstractCompile task has the sourceCompatibility and targetCompatibility inputs. Since the AbstractCompile task is no longer inherited, these inputs are no longer available in Kotlin users' scripts.
- The SourceTask.stableSources input is no longer available, and you should use the sources input. setSource(...) methods are still available.
- All compile tasks now use the libraries input for a list of libraries required for compilation. The KotlinCompile task still has the deprecated Kotlin property classpath, which will be removed in future releases.
- Compile tasks still implement the PatternFilterable interface, which allows the filtering of Kotlin sources. The sourceFilesExtensions input was removed in favor of using PatternFilterable methods.
- The deprecated Gradle destinationDir: File output was replaced with the destinationDirectory: DirectoryProperty output.
- The Kotlin/Native AbstractNativeCompile task now inherits the AbstractKotlinCompileTool base class. This is an initial step toward integrating Kotlin/Native build tools into all the other tools.

Please leave your feedback in this YouTrack ticket.

Statistics of generated files by each annotation processor in kapt

The kotlin-kapt Gradle plugin already reports performance statistics for each processor. Starting with Kotlin 1.7.0, it can also report statistics on the number of generated files for each annotation processor.

This is useful to track if there are unused annotation processors as a part of the build. You can use the generated report to find modules that trigger unnecessary annotation processors and update the modules to prevent that.

Enable the statistics in two steps:

- Set the showProcessorStats flag to true in your build.gradle.kts:

```
kapt {
    showProcessorStats = true
}
```

- Set the kapt.verbose Gradle property to true in your gradle.properties:

```
kapt.verbose=true
```

> You can also enable verbose output via the command line option verbose.

The statistics will appear in the logs with the info level. You'll see the Annotation processor stats: line followed by statistics on the execution time of each annotation processor. After these lines, there will be the Generated files report: line followed by statistics on the number of generated files for each annotation processor. For example:

```
[INFO] Annotation processor stats:
[INFO] org.mapstruct.ap.MappingProcessor: total: 290 ms, init: 1 ms, 3 round(s): 289 ms, 0 ms, 0 ms
[INFO] Generated files report:
[INFO] org.mapstruct.ap.MappingProcessor: total sources: 2, sources per round: 2, 0, 0
```

Please leave your feedback in this YouTrack ticket.

Deprecation of the kotlin.compiler.execution.strategy system property

Kotlin 1.6.20 introduced new properties for defining a Kotlin compiler execution strategy. In Kotlin 1.7.0, a deprecation cycle has started for the old system property kotlin.compiler.execution.strategy in favor of the new properties.

When using the kotlin.compiler.execution.strategy system property, you'll receive a warning. This property will be deleted in future releases. To preserve the old behavior, replace the system property with the Gradle property of the same name. You can do this in gradle.properties, for example:

```
kotlin.compiler.execution.strategy=out-of-process
```

You can also use the compile task property compilerExecutionStrategy. Learn more about this on the Gradle page.

Removal of deprecated options, methods, and plugins

Removal of the useExperimentalAnnotation method

In Kotlin 1.7.0, we completed the deprecation cycle for the useExperimentalAnnotation Gradle method. Use optIn() instead to opt in to using an API in a module.

For example, if your Gradle module is multiplatform:

```
sourceSets {
    all {
        languageSettings.optIn("org.mylibrary.OptInAnnotation")
    }
}
```

Learn more about opt-in requirements in Kotlin.

Removal of deprecated compiler options

We've completed the deprecation cycle for several compiler options:

- The kotlinOptions.jdkHome compiler option was deprecated in 1.5.30 and has been removed in the current release. Gradle builds now fail if they contain this option. We encourage you to use Java toolchains, which have been supported since Kotlin 1.5.30.
- The deprecated noStdlib compiler option has also been removed. The Gradle plugin uses the kotlin.stdlib.default.dependency=true property to control whether the Kotlin standard library is present.

> The compiler arguments -jdkHome and -no-stdlib are still available.

Removal of deprecated plugins

In Kotlin 1.4.0, the kotlin2js and kotlin-dce-plugin plugins were deprecated, and they have been removed in this release. Instead of kotlin2js, use the new org.jetbrains.kotlin.js plugin. Dead code elimination (DCE) works when the Kotlin/JS Gradle plugin is properly configured.

In Kotlin 1.6.0, we changed the deprecation level of the KotlinGradleSubplugin class to ERROR. Developers used this class for writing compiler plugins. In this release, this class has been removed. Use the KotlinCompilerPluginSupportPlugin class instead.

> The best practice is to use Kotlin plugins with versions 1.7.0 and higher throughout your project.

Removal of the deprecated coroutines DSL option and property

We removed the deprecated kotlin.experimental.coroutines Gradle DSL option and the kotlin.coroutines property used in gradle.properties. Now you can just use suspending functions or add the kotlinx.coroutines dependency to your build script.

Learn more about coroutines in the Coroutines guide.

Removal of the type cast in the toolchain extension method

Before Kotlin 1.7.0, you had to do the type cast into the JavaToolchainSpec class when configuring the Gradle toolchain with Kotlin DSL:

```
kotlin {
    jvmToolchain {
        (this as JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
}
```

Now, you can omit the (this as JavaToolchainSpec) part:

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
}
```

Migrating to Kotlin 1.7.0

Install Kotlin 1.7.0

IntelliJ IDEA 2022.1 and Android Studio Chipmunk (212) automatically suggest updating the Kotlin plugin to 1.7.0.

> For IntelliJ IDEA 2022.2, and Android Studio Dolphin (213) or Android Studio Electric Eel (221), the Kotlin plugin 1.7.0 will be delivered with upcoming IntelliJ IDEA and Android Studios updates.

The new command-line compiler is available for download on the GitHub release page.

Migrate existing or start a new project with Kotlin 1.7.0

- To migrate existing projects to Kotlin 1.7.0, change the Kotlin version to 1.7.0 and reimport your Gradle or Maven project. Learn how to update to Kotlin 1.7.0.
- To start a new project with Kotlin 1.7.0, update the Kotlin plugin and run the Project Wizard from File | New | Project.

Compatibility guide for Kotlin 1.7.0

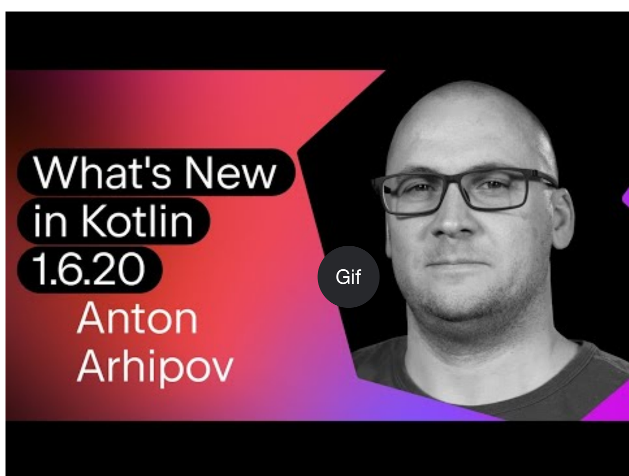
Kotlin 1.7.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of such changes in the Compatibility guide for Kotlin 1.7.0.

What's new in Kotlin 1.6.20

Released: 4 April 2022

Kotlin 1.6.20 reveals previews of the future language features, makes the hierarchical structure the default for multiplatform projects, and brings evolutionary improvements to other components.

You can also find a short overview of the changes in this video:



Watch video online.

Language

In Kotlin 1.6.20, you can try two new language features:

- Prototype of context receivers for Kotlin/JVM
- Definitely non-nullable types

Prototype of context receivers for Kotlin/JVM

> The feature is a prototype available only for Kotlin/JVM. With -Xcontext-receivers enabled, the compiler will produce pre-release binaries that cannot be used in production code. Use context receivers only in your toy projects. We appreciate your feedback in YouTrack.

With Kotlin 1.6.20, you are no longer limited to having one receiver. If you need more, you can make functions, properties, and classes context-dependent (or contextual) by adding context receivers to their declaration. A contextual declaration does the following:

- It requires all declared context receivers to be present in a caller's scope as implicit receivers.
- It brings declared context receivers into its body scope as implicit receivers.

```
interface LoggingContext {
    val log: Logger // This context provides a reference to a logger
}

context(LoggingContext)
fun startBusinessOperation() {
    // You can access the log property since LoggingContext is an implicit receiver
    log.info("Operation has started")
```

```
}

fun test(loggingContext: LoggingContext) {
    with(loggingContext) {
        // You need to have LoggingContext in a scope as an implicit receiver
        // to call startBusinessOperation()
        startBusinessOperation()
    }
}
```

To enable context receivers in your project, use the -Xcontext-receivers compiler option. You can find a detailed description of the feature and its syntax in the KEEP.

Please note that the implementation is a prototype:

- With -Xcontext-receivers enabled, the compiler will produce pre-release binaries that cannot be used in production code
- The IDE support for context receivers is minimal for now

Try the feature in your toy projects and share your thoughts and experience with us in this YouTrack issue. If you run into any problems, please file a new issue.

Definitely non-nullable types

> Definitely non-nullable types are in Beta. They are almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make.

To provide better interoperability when extending generic Java classes and interfaces, Kotlin 1.6.20 allows you to mark a generic type parameter as definitely non-nullable on the use site with the new syntax T & Any. The syntactic form comes from a notation of intersection types and is now limited to a type parameter with nullable upper bounds on the left side of & and non-nullable Any on the right side:

```
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y

fun main() {
    // OK
    elvisLike<String>("", "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String>("", null).length

    // OK
    elvisLike<String?>(null, "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String?>(null, null).length
}
```

Set the language version to 1.7 to enable the feature:

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.7"
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.7'
        }
    }
}
```

Learn more about definitely non-nullable types in the KEEP.

Kotlin/JVM

Kotlin 1.6.20 introduces:

- Compatibility improvements of default methods in JVM interfaces: new @JvmDefaultWithCompatibility annotation for interfaces and compatibility changes in the -Xjvm-default modes
- Support for parallel compilation of a single module in the JVM backend
- Support for callable references to functional interface constructors

New @JvmDefaultWithCompatibility annotation for interfaces

Kotlin 1.6.20 introduces the new annotation @JvmDefaultWithCompatibility: use it along with the -Xjvm-default=all compiler option to create the default method in JVM interface for any non-abstract member in any Kotlin interface.

If there are clients that use your Kotlin interfaces compiled without the -Xjvm-default=all option, they may be binary-incompatible with the code compiled with this option. Before Kotlin 1.6.20, to avoid this compatibility issue, the recommended approach was to use the -Xjvm-default=all-compatibility mode and also the @JvmDefaultWithoutCompatibility annotation for interfaces that didn't need this type of compatibility.

This approach had some disadvantages:

- You could easily forget to add the annotation when a new interface was added.
- Usually there are more interfaces in non-public parts than in the public API, so you end up having this annotation in many places in your code.

Now, you can use the -Xjvm-default=all mode and mark interfaces with the @JvmDefaultWithCompatibility annotation. This allows you to add this annotation to all interfaces in the public API once, and you won't need to use any annotations for new non-public code.

Leave your feedback about this new annotation in this YouTrack ticket.

Compatibility changes in the -Xjvm-default modes

Kotlin 1.6.20 adds the option to compile modules in the default mode (the -Xjvm-default=disable compiler option) against modules compiled with the -Xjvm-default=all or -Xjvm-default=all-compatibility modes. As before, compilations will also be successful if all modules have the -Xjvm-default=all or -Xjvm-default=all-compatibility modes. You can leave your feedback in this YouTrack issue.

Kotlin 1.6.20 deprecates the compatibility and enable modes of the compiler option -Xjvm-default. There are changes in other modes' descriptions regarding the compatibility, but the overall logic remains the same. You can check out the updated descriptions.

For more information about default methods in the Java interop, see the interoperability documentation and this blog post.

Support for parallel compilation of a single module in the JVM backend

> Support for parallel compilation of a single module in the JVM backend is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

We are continuing our work to improve the new JVM IR backend compilation time. In Kotlin 1.6.20, we added the experimental JVM IR backend mode to compile all the files in a module in parallel. Parallel compilation can reduce the total compilation time by up to 15%.

Enable the experimental parallel backend mode with the compiler option -Xbackend-threads. Use the following arguments for this option:

- N is the number of threads you want to use. It should not be greater than your number of CPU cores; otherwise, parallelization stops being effective because of switching context between threads
- 0 to use a separate thread for each CPU core

Gradle can run tasks in parallel, but this type of parallelization doesn't help a lot when a project (or a major part of a project) is just one big task from Gradle's perspective. If you have a very big monolithic module, use parallel compilation to compile more quickly. If your project consists of lots of small modules and has a build parallelized by Gradle, adding another layer of parallelization may hurt performance because of context switching.

Parallel compilation has some constraints:

- It doesn't work with kapt because kapt disables the IR backend
- It requires more JVM heap by design. The amount of heap is proportional to the number of threads

Support for callable references to functional interface constructors

Support for callable references to functional interface constructors is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Support for callable references to functional interface constructors adds a source-compatible way to migrate from an interface with a constructor function to a functional interface.

Consider the following code:

```
interface Printer {
    fun print()
}

fun Printer(block: () -> Unit): Printer = object : Printer { override fun print() = block() }
```

With callable references to functional interface constructors enabled, this code can be replaced with just a functional interface declaration:

```
fun interface Printer {
    fun print()
}
```

Its constructor will be created implicitly, and any code using the ::Printer function reference will compile. For example:

```
documentsStorage.addPrinter(::Printer)
```

Preserve the binary compatibility by marking the legacy function Printer with the @Deprecated annotation with DeprecationLevel.HIDDEN:

```
@Deprecated(message = "Your message about the deprecation", level = DeprecationLevel.HIDDEN)
fun Printer(...) {...}
```

Use the compiler option -XXLanguage:+KotlinFunInterfaceConstructorReference to enable this feature.

Kotlin/Native

Kotlin/Native 1.6.20 marks continued development of its new components. We've taken another step toward consistent experience with Kotlin on other platforms:

- An update on the new memory manager
- Concurrent implementation for the sweep phase in new memory manager
- Instantiation of annotation classes
- Interop with Swift async/await: returning Swift's Void instead of KotlinUnit
- Better stack traces with libbacktrace
- Support for standalone Android executables
- Performance improvements
- Improved error handling during cinterop modules import
- Support for Xcode 13 libraries

An update on the new memory manager

> The new Kotlin/Native memory manager is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback on it in YouTrack.

With Kotlin 1.6.20, you can try the Alpha version of the new Kotlin/Native memory manager. It eliminates the differences between the JVM and Native platforms to provide a consistent developer experience in multiplatform projects. For example, you'll have a much easier time creating new cross-platform mobile applications that work on both Android and iOS.

The new Kotlin/Native memory manager lifts restrictions on object-sharing between threads. It also provides leak-free concurrent programming primitives that are safe and don't require any special management or annotations.

The new memory manager will become the default in future versions, so we encourage you to try it now. Check out our blog post to learn more about the new memory manager and explore demo projects, or jump right to the migration instructions to try it yourself.

Try using the new memory manager on your projects to see how it works and share feedback in our issue tracker, YouTrack.

Concurrent implementation for the sweep phase in new memory manager

If you have already switched to our new memory manager, which was announced in Kotlin 1.6, you might notice a huge execution time improvement: our benchmarks show 35% improvement on average. Starting with 1.6.20, there is also a concurrent implementation for the sweep phase available for the new memory manager. This should also improve the performance and decrease the duration of garbage collector pauses.

To enable the feature for the new Kotlin/Native memory manager, pass the following compiler option:

```
-Xgc=cms
```

Feel free to share your feedback on the new memory manager performance in this YouTrack issue.

Instantiation of annotation classes

In Kotlin 1.6.0, instantiation of annotation classes became Stable for Kotlin/JVM and Kotlin/JS. The 1.6.20 version delivers support for Kotlin/Native.

Learn more about instantiation of annotation classes.

Interop with Swift async/await: returning Void instead of KotlinUnit

> Concurrency interoperability with Swift async/await is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

We've continued working on the experimental interop with Swift's async/await (available since Swift 5.5). Kotlin 1.6.20 differs from previous versions in the way it works with suspend functions with the Unit return type.

Previously, such functions were presented in Swift as async functions returning KotlinUnit. However, the proper return type for them is Void, similar to non-suspending functions.

To avoid breaking the existing code, we're introducing a Gradle property that makes the compiler translate Unit-returning suspend functions to async Swift with the Void return type:

```
# gradle.properties
kotlin.native.binary.unitSuspendFunctionObjCExport=proper
```

We plan to make this behavior the default in future Kotlin releases.

Better stack traces with libbacktrace

> Using libbacktrace for resolving source locations is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin/Native is now able to produce detailed stack traces with file locations and line numbers for better debugging of linux* (except linuxMips32 and linuxMipsel32) and androidNative* targets.

This feature uses the libbacktrace library under the hood. Take a look at the following code to see an example of the difference:

```
fun main() = bar()
fun bar() = baz()
inline fun baz() {
    error("")
}
```

- Before 1.6.20:

Uncaught Kotlin exception: kotlin.IllegalStateException: at 0 example.kexe 0x227190 kfun:kotlin.Throwable#<init>(kotlin.String?){} + 96 at 1 example.kexe 0x221e4c kfun:kotlin.Exception#<init>(kotlin.String?){} + 92 at 2 example.kexe 0x221f4c kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 92 at 3 example.kexe 0x22234c kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} + 92 at 4 example.kexe 0x25d708 kfun:#bar(){} + 104 at 5 example.kexe 0x25d68c kfun:#main(){} + 12

- 1.6.20 with libbacktrace:

Uncaught Kotlin exception: kotlin.IllegalStateException: at 0 example.kexe 0x229550 kfun:kotlin.Throwable#<init>(kotlin.String?){} + 96 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37) at 1 example.kexe 0x22420c kfun:kotlin.Exception#<init>(kotlin.String?){} + 92 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44) at 2 example.kexe 0x22430c kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 92 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44) at 3 example.kexe 0x22470c kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} + 92 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44) at 4 example.kexe 0x25fac8 kfun:#bar(){} + 104 [inlined] (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/libraries/stdlib/src/kotlin/util/Preconditions.kt:143:56) at 5 example.kexe 0x25fac8 kfun:#bar(){} + 104 [inlined] (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:4:5) at 6 example.kexe 0x25fac8 kfun:#bar(){} + 104 (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:13) at 7 example.kexe 0x25fa4c kfun:#main(){} + 12 (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14)

On Apple targets, which already had file locations and line numbers in stack traces, libbacktrace provides more details for inline function calls:

- Before 1.6.20:

Uncaught Kotlin exception: kotlin.IllegalStateException: at 0 example.kexe 0x10a85a8f8 kfun:kotlin.Throwable#<init>(kotlin.String?){} + 88 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37) at 1 example.kexe 0x10a855846 kfun:kotlin.Exception#<init>(kotlin.String?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44) at 2 example.kexe 0x10a855936 kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44) at 3 example.kexe 0x10a855c86 kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44) at 4 example.kexe 0x10a8489a5 kfun:#bar(){} + 117 (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:1) at 5 example.kexe 0x10a84891c kfun:#main(){} + 12 (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14) ...

- 1.6.20 with libbacktrace:

Uncaught Kotlin exception: kotlin.IllegalStateException: at 0 example.kexe 0x10669bc88 kfun:kotlin.Throwable#<init>(kotlin.String?){} + 88 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37) at 1 example.kexe 0x106696bd6 kfun:kotlin.Exception#<init>(kotlin.String?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44) at 2 example.kexe 0x106696cc6 kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44) at 3 example.kexe 0x106697016 kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44) at 4 example.kexe 0x106689d35 kfun:#bar(){} + 117 [inlined] (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/libraries/stdlib/src/kotlin/util/Preconditions.kt:143:56) >> at 5 example.kexe 0x106689d35 kfun:#bar(){} + 117 [inlined] (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:4:5) at 6 example.kexe 0x106689d35 kfun:#bar(){} + 117 (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:13) at 7 example.kexe 0x106689cac kfun:#main(){} + 12 (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14) ...

To produce better stack traces with libbacktrace, add the following line to gradle.properties:

```
# gradle.properties
kotlin.native.binary.sourceInfoType=libbacktrace
```

Please tell us how debugging Kotlin/Native with libbacktrace works for you in this YouTrack issue.

Support for standalone Android executables

Previously, Android Native executables in Kotlin/Native were not actually executables but shared libraries that you could use as a NativeActivity. Now there's an option to generate standard executables for Android Native targets.

For that, in the build.gradle(.kts) part of your project, configure the executable block of your androidNative target. Add the following binary option:

```
kotlin {
    androidNativeX64("android") {
        binaries {
            executable {
                binaryOptions["androidProgramType"] = "standalone"
            }
        }
    }
}
```

Note that this feature will become the default in Kotlin 1.7.0. If you want to preserve the current behavior, use the following setting:

```
binaryOptions["androidProgramType"] = "nativeActivity"
```

Thanks to Mattia Iavarone for the implementation!

Performance improvements

We are working hard on Kotlin/Native to speed up the compilation process and improve your developing experience.

Kotlin 1.6.20 brings some performance updates and bug fixes that affect the LLVM IR that Kotlin generates. According to the benchmarks on our internal projects, we achieved the following performance boosts on average:

- 15% reduction in execution time
- 20% reduction in the code size of both release and debug binaries
- 26% reduction in the compilation time of release binaries

These changes also provide a 10% reduction in compilation time for a debug binary on a large internal project.

To achieve this, we've implemented static initialization for some of the compiler-generated synthetic objects, improved the way we structure LLVM IR for every function, and optimized the compiler caches.

Improved error handling during cinterop modules import

This release introduces improved error handling for cases where you import an Objective-C module using the cinterop tool (as is typical for CocoaPods pods). Previously, if you got an error while trying to work with an Objective-C module (for instance, when dealing with a compilation error in a header), you received an uninformative error message, such as fatal error: could not build module $name. We expanded upon this part of the cinterop tool, so you'll get an error message with an extended description.

Support for Xcode 13 libraries

Libraries delivered with Xcode 13 have full support as of this release. Feel free to access them from anywhere in your Kotlin code.

Kotlin Multiplatform

1.6.20 brings the following notable updates to Kotlin Multiplatform:

- Hierarchical structure support is now default for all new multiplatform projects
- Kotlin CocoaPods Gradle plugin received several useful features for CocoaPods integration

Hierarchical structure support for multiplatform projects

Kotlin 1.6.20 comes with hierarchical structure support enabled by default. Since introducing it in Kotlin 1.4.0, we've significantly improved the frontend and made IDE import stable.

Previously, there were two ways to add code in a multiplatform project. The first was to insert it in a platform-specific source set, which is limited to one target and can't be reused by other platforms. The second is to use a common source set shared across all the platforms that are currently supported by Kotlin.

Now you can share source code among several similar native targets that reuse a lot of the common logic and third-party APIs. The technology will provide the correct default dependencies and find the exact API available in the shared code. This eliminates a complex build setup and having to use workarounds to get IDE support for sharing source sets among native targets. It also helps prevent unsafe API usages meant for a different target.

The technology will come in handy for library authors, too, as a hierarchical project structure allows them to publish and consume libraries with common APIs for a subset of targets.

By default, libraries published with the hierarchical project structure are compatible only with hierarchical structure projects.

Better code-sharing in your project

Without hierarchical structure support, there is no straightforward way to share code across some but not all Kotlin targets. One popular example is sharing code across all iOS targets and having access to iOS-specific dependencies, like Foundation.

Thanks to the hierarchical project structure support, you can now achieve this out of the box. In the new structure, source sets form a hierarchy. You can use platform-specific language features and dependencies available for each target that a given source set compiles to.

For example, consider a typical multiplatform project with two targets — iosArm64 and iosX64 for iOS devices and simulators. The Kotlin tooling understands that both targets have the same function and allows you to access that function from the intermediate source set, iosMain.

iOS hierarchy example

The Kotlin toolchain provides the correct default dependencies, like Kotlin/Native stdlib or native libraries. Moreover, Kotlin tooling will try its best to find exactly the API surface area available in the shared code. This prevents such cases as, for example, the use of a macOS-specific function in code shared for Windows.

More opportunities for library authors

When a multiplatform library is published, the API of its intermediate source sets is now properly published alongside it, making it available for consumers. Again, the Kotlin toolchain will automatically figure out the API available in the consumer source set while carefully watching out for unsafe usages, like using an API meant for the JVM in JS code. Learn more about sharing code in libraries.

Configuration and setup

Starting with Kotlin 1.6.20, all your new multiplatform projects will have a hierarchical project structure. No additional setup is required.

- If you've already turned it on manually, you can remove the deprecated options from gradle.properties:

```
# gradle.properties
kotlin.mpp.enableGranularSourceSetsMetadata=true
kotlin.native.enableDependencyPropagation=false // or 'true', depending on your previous setup
```

- For Kotlin 1.6.20, we recommend using Android Studio 2021.1.1 (Bumblebee) or later to get the best experience.

- You can also opt out. To disable hierarchical structure support, set the following options in gradle.properties:

```
# gradle.properties
kotlin.mpp.hierarchicalStructureSupport=false
```

Leave your feedback

This is a significant change to the whole ecosystem. We would appreciate your feedback to help make it even better.

Try it now and report any difficulties you encounter to our issue tracker.

Kotlin CocoaPods Gradle plugin

To simplify CocoaPods integration, Kotlin 1.6.20 delivers the following features:

- The CocoaPods plugin now has tasks that build XCFrameworks with all registered targets and generate the Podspec file. This can be useful when you don't want to integrate with Xcode directly, but you want to build artifacts and deploy them to your local CocoaPods repository.

  Learn more about building XCFrameworks.

- If you use CocoaPods integration in your projects, you're used to specifying the required Pod version for the entire Gradle project. Now you have more options:

  - Specify the Pod version directly in the cocoapods block
  - Continue using a Gradle project version

  If none of these properties is configured, you'll get an error.

- You can now configure the CocoaPod name in the cocoapods block instead of changing the name of the whole Gradle project.

- The CocoaPods plugin introduces a new extraSpecAttributes property, which you can use to configure properties in a Podspec file that were previously hard-coded, like libraries or vendored_frameworks.

```
kotlin {
    cocoapods {
        version = "1.0"
        name = "MyCocoaPod"
        extraSpecAttributes["social_media_url"] = 'https://twitter.com/kotlin'
        extraSpecAttributes["vendored_frameworks"] = 'CustomFramework.xcframework'
        extraSpecAttributes["libraries"] = 'xml'
    }
}
```

See the full Kotlin CocoaPods Gradle plugin DSL reference.

Kotlin/JS

Kotlin/JS improvements in 1.6.20 mainly affect the IR compiler:

- Incremental compilation for development binaries (IR)
- Lazy initialization of top-level properties by default (IR)
- Separate JS files for project modules by default (IR)
- Char class optimization (IR)
- Export improvements (both IR and legacy backends)
- @AfterTest guarantees for asynchronous tests

Incremental compilation for development binaries with IR compiler

To make Kotlin/JS development with the IR compiler more efficient, we're introducing a new incremental compilation mode.

When building development binaries with the compileDevelopmentExecutableKotlinJs Gradle task in this mode, the compiler caches the results of previous compilations on the module level. It uses the cached compilation results for unchanged source files during subsequent compilations, making them complete more quickly, especially with small changes. Note that this improvement exclusively targets the development process (shortening the edit-build-debug cycle) and doesn't affect the building of production artifacts.

To enable incremental compilation for development binaries, add the following line to the project's gradle.properties:

```
# gradle.properties
kotlin.incremental.js.ir=true // false by default
```

In our test projects, the new mode made incremental compilation up to 30% faster. However, the clean build in this mode became slower because of the need to create and populate the caches.

Please tell us what you think of using incremental compilation with your Kotlin/JS projects in this YouTrack issue.

Lazy initialization of top-level properties by default with IR compiler

In Kotlin 1.4.30, we presented a prototype of lazy initialization of top-level properties in the JS IR compiler. By eliminating the need to initialize all properties when the application launches, lazy initialization reduces the startup time. Our measurements showed about a 10% speed-up on a real-life Kotlin/JS application.

Now, having polished and properly tested this mechanism, we're making lazy initialization the default for top-level properties in the IR compiler.

```
// lazy initialization
val a = run {
    val result = // intensive computations
        println(result)
    result
} // run is executed upon the first usage of the variable
```

If for some reason you need to initialize a property eagerly (upon the application start), mark it with the @EagerInitialization annotation.

Separate JS files for project modules by default with IR compiler

Previously, the JS IR compiler offered an ability to generate separate .js files for project modules. This was an alternative to the default option – a single .js file for the whole project. This file might be too large and inconvenient to use, because whenever you want to use a function from your project, you have to include the entire JS file as a dependency. Having multiple files adds flexibility and decreases the size of such dependencies. This feature was available with the -Xir-per-module compiler option.

Starting from 1.6.20, the JS IR compiler generates separate .js files for project modules by default.

Compiling the project into a single .js file is now available with the following Gradle property:

```
# gradle.properties
kotlin.js.ir.output.granularity=whole-program // `per-module` is the default
```

In previous releases, the experimental per-module mode (available via the -Xir-per-module=true flag) invoked main() functions in each module. This is inconsistent with the regular single .js mode. Starting with 1.6.20, the main() function will be invoked in the main module only in both cases. If you do need to run some code when a module is loaded, you can use top-level properties annotated with the @EagerInitialization annotation. See Lazy initialization of top-level properties by default (IR).

Char class optimization

The Char class is now handled by the Kotlin/JS compiler without introducing boxing (similar to inline classes). This speeds up operations on chars in Kotlin/JS code.

Aside from the performance improvement, this changes the way Char is exported to JavaScript: it's now translated to Number.

Improvements to export and TypeScript declaration generation

Kotlin 1.6.20 is bringing multiple fixes and improvements to the export mechanism (the @JsExport annotation), including the generation of TypeScript declarations (.d.ts). We've added the ability to export interfaces and enums, and we've fixed the export behavior in some corner cases that were reported to us previously. For more details, see the list of export improvements in YouTrack.

Learn more about using Kotlin code from JavaScript.

@AfterTest guarantees for asynchronous tests

Kotlin 1.6.20 makes @AfterTest functions work properly with asynchronous tests on Kotlin/JS. If a test function's return type is statically resolved to Promise, the compiler now schedules the execution of the @AfterTest function to the corresponding then() callback.

Security

Kotlin 1.6.20 introduces a couple of features to improve the security of your code:

- Using relative paths in klibs
- Persisting yarn.lock for Kotlin/JS Gradle projects
- Installation of npm dependencies with --ignore-scripts by default

Using relative paths in klibs

A library in klib format contains a serialized IR representation of source files, which also includes their paths for generating proper debug information. Before Kotlin 1.6.20, stored file paths were absolute. Since the library author may not want to share absolute paths, the 1.6.20 version comes with an alternative option.

If you are publishing a klib and want to use only relative paths of source files in the artifact, you can now pass the -Xklib-relative-path-base compiler option with one or multiple base paths of source files:

Kotlin

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinCompile::class).configureEach {
    // $base is a base path of source files
    kotlinOptions.freeCompilerArgs += "-Xklib-relative-path-base=$base"
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinCompile).configureEach {
    kotlinOptions {
        // $base is a base path of source files
        freeCompilerArgs += "-Xklib-relative-path-base=$base"
    }
}
```

Persisting yarn.lock for Kotlin/JS Gradle projects

> The feature was backported to Kotlin 1.6.10.

The Kotlin/JS Gradle plugin now provides an ability to persist the yarn.lock file, making it possible to lock the versions of the npm dependencies for your project without additional Gradle configuration. The feature brings changes to the default project structure by adding the auto-generated kotlin-js-store directory to the project root. It holds the yarn.lock file inside.

We strongly recommend committing the kotlin-js-store directory and its contents to your version control system. Committing lockfiles to your version control system is a recommended practice because it ensures your application is being built with the exact same dependency tree on all machines, regardless of whether those are development environments on other machines or CI/CD services. Lockfiles also prevent your npm dependencies from being silently updated when a project is checked out on a new machine, which is a security concern.

Tools like Dependabot can also parse the yarn.lock files of your Kotlin/JS projects, and provide you with warnings if any npm package you depend on is compromised.

If needed, you can change both directory and lockfile names in the build script:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().lockFileDirectory =
        project.rootDir.resolve("my-kotlin-js-store")
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().lockFileName = "my-yarn.lock"
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFileDirectory =
```

```
        file("my-kotlin-js-store")
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFileName = 'my-yarn.lock'
}
```

Changing the name of the lockfile may cause dependency inspection tools to no longer pick up the file.

Installation of npm dependencies with --ignore-scripts by default

The feature was backported to Kotlin 1.6.10.

The Kotlin/JS Gradle plugin now prevents the execution of lifecycle scripts during the installation of npm dependencies by default. The change is aimed at reducing the likelihood of executing malicious code from compromised npm packages.

To roll back to the old configuration, you can explicitly enable lifecycle scripts execution by adding the following lines to build.gradle(.kts):

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().ignoreScripts = false
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).ignoreScripts = false
}
```

Learn more about npm dependencies of a Kotlin/JS Gradle project.

Gradle

Kotlin 1.6.20 brings the following changes for the Kotlin Gradle Plugin:

- New properties kotlin.compiler.execution.strategy and compilerExecutionStrategy for defining a Kotlin compiler execution strategy
- Deprecation of the options kapt.use.worker.api, kotlin.experimental.coroutines, and kotlin.coroutines
- Removal of the kotlin.parallel.tasks.in.project build option

Properties for defining Kotlin compiler execution strategy

Before Kotlin 1.6.20, you used the system property -Dkotlin.compiler.execution.strategy to define a Kotlin compiler execution strategy. This property might have been inconvenient in some cases. Kotlin 1.6.20 introduces a Gradle property with the same name, kotlin.compiler.execution.strategy, and the compile task property compilerExecutionStrategy.

The system property still works, but it will be removed in future releases.

The current priority of properties is the following:

- The task property compilerExecutionStrategy takes priority over the system property and the Gradle property kotlin.compiler.execution.strategy.
- The Gradle property takes priority over the system property.

There are three compiler execution strategies that you can assign to these properties:

Strategy	Where Kotlin compiler is executed	Incremental compilation	Other characteristics

Strategy	Where Kotlin compiler is executed	Incremental compilation	Other characteristics
Daemon	Inside its own daemon process	Yes	The default strategy. Can be shared between different Gradle daemons
In process	Inside the Gradle daemon process	No	May share the heap with the Gradle daemon
Out of process	In a separate process for each call	No	—

Accordingly, the available values for kotlin.compiler.execution.strategy properties (both system and Gradle's) are:

1. daemon (default)
2. in-process
3. out-of-process

Use the Gradle property kotlin.compiler.execution.strategy in gradle.properties:

```
# gradle.properties
kotlin.compiler.execution.strategy=out-of-process
```

The available values for the compilerExecutionStrategy task property are:

1. org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.DAEMON (default)
2. org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.IN_PROCESS
3. org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.OUT_OF_PROCESS

Use the task property compilerExecutionStrategy in the build.gradle.kts build script:

```
import org.jetbrains.kotlin.gradle.dsl.KotlinCompile
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType<KotlinCompile>().configureEach {
    compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN_PROCESS)
}
```

Please leave your feedback in this YouTrack task.

Deprecation of build options for kapt and coroutines

In Kotlin 1.6.20, we changed deprecation levels of the properties:

- We deprecated the ability to run kapt via the Kotlin daemon with kapt.use.worker.api – now it produces a warning to Gradle's output. By default, kapt has been using Gradle workers since the 1.3.70 release, and we recommend sticking to this method.

  We are going to remove the option kapt.use.worker.api in future releases.

- We deprecated the kotlin.experimental.coroutines Gradle DSL option and the kotlin.coroutines property used in gradle.properties. Just use suspending functions or add the kotlinx.coroutines dependency to your build.gradle(.kts) file.

  Learn more about coroutines in the Coroutines guide.

Removal of the kotlin.parallel.tasks.in.project build option

In Kotlin 1.5.20, we announced the deprecation of the build option kotlin.parallel.tasks.in.project. This option has been removed in Kotlin 1.6.20.

Depending on the project, parallel compilation in the Kotlin daemon may require more memory. To reduce memory consumption, increase the heap size for the Kotlin daemon.

Learn more about the currently supported compiler options in the Kotlin Gradle plugin.

What's new in Kotlin 1.6.0

Released: 16 November 2021

Kotlin 1.6.0 introduces new language features, optimizations and improvements to existing features, and a lot of improvements to the Kotlin standard library.

You can also find an overview of the changes in the release blog post.

Language

Kotlin 1.6.0 brings stabilization to several language features introduced for preview in the previous 1.5.30 release:

- Stable exhaustive when statements for enum, sealed and Boolean subjects
- Stable suspending functions as supertypes
- Stable suspend conversions
- Stable instantiation of annotation classes

It also includes various type inference improvements and support for annotations on class type parameters:

- Improved type inference for recursive generic types
- Changes to builder inference
- Support for annotations on class type parameters

Stable exhaustive when statements for enum, sealed, and Boolean subjects

An exhaustive when statement contains branches for all possible types or values of its subject, or for some types plus an else branch. It covers all possible cases, making your code safer.

We will soon prohibit non-exhaustive when statements to make the behavior consistent with when expressions. To ensure smooth migration, Kotlin 1.6.0 reports warnings about non-exhaustive when statements with an enum, sealed, or Boolean subject. These warnings will become errors in future releases.

```
sealed class Contact {
    data class PhoneCall(val number: String) : Contact()
    data class TextMessage(val number: String) : Contact()
}

fun Contact.messageCost(): Int =
    when(this) { // Error: 'when' expression must be exhaustive
        is Contact.PhoneCall -> 42
    }

fun sendMessage(contact: Contact, message: String) {
    // Starting with 1.6.0

    // Warning: Non exhaustive 'when' statements on Boolean will be
    // prohibited in 1.7, add 'false' branch or 'else' branch instead
    when(message.isEmpty()) {
        true -> return
    }
    // Warning: Non exhaustive 'when' statements on sealed class/interface will be
    // prohibited in 1.7, add 'is TextMessage' branch or 'else' branch instead
    when(contact) {
        is Contact.PhoneCall -> TODO()
    }
}
```

See this YouTrack ticket for a more detailed explanation of the change and its effects.

Stable suspending functions as supertypes

Implementation of suspending functional types has become Stable in Kotlin 1.6.0. A preview was available in 1.5.30.

The feature can be useful when designing APIs that use Kotlin coroutines and accept suspending functional types. You can now streamline your code by enclosing the desired behavior in a separate class that implements a suspending functional type.

```
class MyClickAction : suspend () -> Unit {
    override suspend fun invoke() { TODO() }
}

fun launchOnClick(action: suspend () -> Unit) {}
```

You can use an instance of this class where only lambdas and suspending function references were allowed previously: launchOnClick(MyClickAction()).

There are currently two limitations coming from implementation details:

- You can't mix ordinary functional types and suspending ones in the list of supertypes.
- You can't use multiple suspending functional supertypes.

Stable suspend conversions

Kotlin 1.6.0 introduces Stable conversions from regular to suspending functional types. Starting from 1.4.0, the feature supported functional literals and callable references. With 1.6.0, it works with any form of expression. As a call argument, you can now pass any expression of a suitable regular functional type where suspending is expected. The compiler will perform an implicit conversion automatically.

```
fun getSuspending(suspending: suspend () -> Unit) {}

fun suspending() {}

fun test(regular: () -> Unit) {
    getSuspending { }           // OK
    getSuspending(::suspending) // OK
    getSuspending(regular)      // OK
}
```

Stable instantiation of annotation classes

Kotlin 1.5.30 introduced experimental support for instantiation of annotation classes on the JVM platform. With 1.6.0, the feature is available by default both for Kotlin/JVM and Kotlin/JS.

Learn more about instantiation of annotation classes in this KEEP.

Improved type inference for recursive generic types

Kotlin 1.5.30 introduced an improvement to type inference for recursive generic types, which allowed their type arguments to be inferred based only on the upper bounds of the corresponding type parameters. The improvement was available with the compiler option. In version 1.6.0 and later, it is enabled by default.

```
// Before 1.5.30
val containerA = PostgreSQLContainer<Nothing>(DockerImageName.parse("postgres:13-alpine")).apply {
  withDatabaseName("db")
  withUsername("user")
  withPassword("password")
  withInitScript("sql/schema.sql")
}

// With compiler option in 1.5.30 or by default starting with 1.6.0
val containerB = PostgreSQLContainer(DockerImageName.parse("postgres:13-alpine"))
  .withDatabaseName("db")
  .withUsername("user")
  .withPassword("password")
  .withInitScript("sql/schema.sql")
```

Changes to builder inference

Builder inference is a type inference flavor which is useful when calling generic builder functions. It can infer the type arguments of a call with the help of type information from calls inside its lambda argument.

We're making multiple changes that are bringing us closer to fully stable builder inference. Starting with 1.6.0:

- You can make calls returning an instance of a not yet inferred type inside a builder lambda without specifying the -Xunrestricted-builder-inference compiler

option introduced in 1.5.30.

- With -Xenable-builder-inference, you can write your own builders without applying the @BuilderInference annotation.

> Note that clients of these builders will need to specify the same -Xenable-builder-inference compiler option.

- With the -Xenable-builder-inference, builder inference automatically activates if a regular type inference cannot get enough information about a type.

Learn how to write custom generic builders.

Support for annotations on class type parameters

Support for annotations on class type parameters looks like this:

```
@Target(AnnotationTarget.TYPE_PARAMETER)
annotation class BoxContent

class Box<@BoxContent T> {}
```

Annotations on all type parameters are emitted into JVM bytecode so annotation processors are able to use them.

For the motivating use case, read this YouTrack ticket.

Learn more about annotations.

Supporting previous API versions for a longer period

Starting with Kotlin 1.6.0, we will support development for three previous API versions instead of two, along with the current stable one. Currently, we support versions 1.3, 1.4, 1.5, and 1.6.

Kotlin/JVM

For Kotlin/JVM, starting with 1.6.0, the compiler can generate classes with a bytecode version corresponding to JVM 17. The new language version also includes optimized delegated properties and repeatable annotations, which we had on the roadmap:

- Repeatable annotations with runtime retention for 1.8 JVM target
- Optimize delegated properties which call get/set on the given KProperty instance

Repeatable annotations with runtime retention for 1.8 JVM target

Java 8 introduced repeatable annotations, which can be applied multiple times to a single code element. The feature requires two declarations to be present in the Java code: the repeatable annotation itself marked with @java.lang.annotation.Repeatable and the containing annotation to hold its values.

Kotlin also has repeatable annotations, but requires only @kotlin.annotation.Repeatable to be present on an annotation declaration to make it repeatable. Before 1.6.0, the feature supported only SOURCE retention and was incompatible with Java's repeatable annotations. Kotlin 1.6.0 removes these limitations. @kotlin.annotation.Repeatable now accepts any retention and makes the annotation repeatable both in Kotlin and Java. Java's repeatable annotations are now also supported from the Kotlin side.

While you can declare a containing annotation, it's not necessary. For example:

- If an annotation @Tag is marked with @kotlin.annotation.Repeatable, the Kotlin compiler automatically generates a containing annotation class under the name of @Tag.Container:

```
@Repeatable
annotation class Tag(val name: String)

// The compiler generates @Tag.Container containing annotation
```

- To set a custom name for a containing annotation, apply the @kotlin.jvm.JvmRepeatable meta-annotation and pass the explicitly declared containing annotation class as an argument:

```kotlin
@JvmRepeatable(Tags::class)
annotation class Tag(val name: String)

annotation class Tags(val value: Array<Tag>)
```

Kotlin reflection now supports both Kotlin's and Java's repeatable annotations via a new function, KAnnotatedElement.findAnnotations().

Learn more about Kotlin repeatable annotations in this KEEP.

Optimize delegated properties which call get/set on the given KProperty instance

We optimized the generated JVM bytecode by omitting the $delegate field and generating immediate access to the referenced property.

For example, in the following code

```kotlin
class Box<T> {
    private var impl: T = ...

    var content: T by ::impl
}
```

Kotlin no longer generates the field content$delegate. Property accessors of the content variable invoke the impl variable directly, skipping the delegated property's getValue/setValue operators and thus avoiding the need for the property reference object of the KProperty type.

Thanks to our Google colleagues for the implementation!

Learn more about delegated properties.

Kotlin/Native

Kotlin/Native is receiving multiple improvements and component updates, some of them in the preview state:

- Preview of the new memory manager
- Support for Xcode 13
- Compilation of Windows targets on any host
- LLVM and linker updates
- Performance improvements
- Unified compiler plugin ABI with JVM and JS IR backends
- Detailed error messages for klib linkage failures
- Reworked unhandled exception handling API

Preview of the new memory manager

> The new Kotlin/Native memory manager is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

With Kotlin 1.6.0, you can try the development preview of the new Kotlin/Native memory manager. It moves us closer to eliminating the differences between the JVM and Native platforms to provide a consistent developer experience in multiplatform projects.

One of the notable changes is the lazy initialization of top-level properties, like in Kotlin/JVM. A top-level property gets initialized when a top-level property or function from the same file is accessed for the first time. This mode also includes global interprocedural optimization (enabled only for release binaries), which removes redundant initialization checks.

We've recently published a blog post about the new memory manager. Read it to learn about the current state of the new memory manager and find some demo projects, or jump right to the migration instructions to try it yourself. Please check how the new memory manager works on your projects and share feedback in our issue tracker, YouTrack.

Support for Xcode 13

Kotlin/Native 1.6.0 supports Xcode 13 – the latest version of Xcode. Feel free to update your Xcode and continue working on your Kotlin projects for Apple operating systems.

> New libraries added in Xcode 13 aren't available for use in Kotlin 1.6.0, but we're going to add support for them in upcoming versions.

Compilation of Windows targets on any host

Starting from 1.6.0, you don't need a Windows host to compile the Windows targets mingwX64 and mingwX86. They can be compiled on any host that supports Kotlin/Native.

LLVM and linker updates

We've reworked the LLVM dependency that Kotlin/Native uses under the hood. This brings various benefits, including:

- Updated LLVM version to 11.1.0.
- Decreased dependency size. For example, on macOS it's now about 300 MB instead of 1200 MB in the previous version.
- Excluded dependency on the ncurses5 library that isn't available in modern Linux distributions.

In addition to the LLVM update, Kotlin/Native now uses the LLD linker (a linker from the LLVM project) for MingGW targets. It provides various benefits over the previously used ld.bfd linker, and will allow us to improve runtime performance of produced binaries and support compiler caches for MinGW targets. Note that LLD requires import libraries for DLL linkage. Learn more in this Stack Overflow thread.

Performance improvements

Kotlin/Native 1.6.0 delivers the following performance improvements:

- Compilation time: compiler caches are enabled by default for linuxX64 and iosArm64 targets. This speeds up most compilations in debug mode (except the first one). Measurements showed about a 200% speed increase on our test projects. The compiler caches have been available for these targets since Kotlin 1.5.0 with additional Gradle properties; you can remove them now.
- Runtime: iterating over arrays with for loops is now up to 12% faster thanks to optimizations in the produced LLVM code.

Unified compiler plugin ABI with JVM and JS IR backends

> The option to use the common IR compiler plugin ABI for Kotlin/Native is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

In previous versions, authors of compiler plugins had to provide separate artifacts for Kotlin/Native because of the differences in the ABI.

Starting from 1.6.0, the Kotlin Multiplatform Gradle plugin is able to use the embeddable compiler jar – the one used for the JVM and JS IR backends – for Kotlin/Native. This is a step toward unification of the compiler plugin development experience, as you can now use the same compiler plugin artifacts for Native and other supported platforms.

This is a preview version of such support, and it requires an opt-in. To start using generic compiler plugin artifacts for Kotlin/Native, add the following line to gradle.properties: kotlin.native.useEmbeddableCompilerJar=true.

We're planning to use the embeddable compiler jar for Kotlin/Native by default in the future, so it's vital for us to hear how the preview works for you.

If you are an author of a compiler plugin, please try this mode and check if it works for your plugin. Note that depending on your plugin's structure, migration steps may be required. See this YouTrack issue for migration instructions and leave your feedback in the comments.

Detailed error messages for klib linkage failures

The Kotlin/Native compiler now provides detailed error messages for klib linkage errors. The messages now have clear error descriptions, and they also include information about possible causes and ways to fix them.

For example:

- 1.5.30:

```
e: java.lang.IllegalStateException: IrTypeAliasSymbol expected: Unbound public symbol for public
kotlinx.coroutines/CancellationException|null[0]
<stack trace>
```

- 1.6.0:

```
e: The symbol of unexpected type encountered during IR deserialization: IrClassPublicSymbolImpl,
kotlinx.coroutines/CancellationException|null[0].
IrTypeAliasSymbol is expected.

This could happen if there are two libraries, where one library was compiled against the different version of the other library
than the one currently used in the project.
Please check that the project configuration is correct and has consistent versions of dependencies.

The list of libraries that depend on "org.jetbrains.kotlinx:kotlinx-coroutines-core (org.jetbrains.kotlinx:kotlinx-coroutines-
core-macosx64)" and may lead to conflicts:
<list of libraries and potential version mismatches>

Project dependencies:
<dependencies tree>
```

Reworked unhandled exception handling API

We've unified the processing of unhandled exceptions throughout the Kotlin/Native runtime and exposed the default processing as the function processUnhandledException(throwable: Throwable) for use by custom execution environments, like kotlinx.coroutines. This processing is also applied to exceptions that escape operation in Worker.executeAfter(), but only for the new memory manager.

API improvements also affected the hooks that have been set by setUnhandledExceptionHook(). Previously such hooks were reset after the Kotlin/Native runtime called the hook with an unhandled exception, and the program would always terminate right after. Now these hooks may be used more than once, and if you want the program to always terminate on an unhandled exception, either do not set an unhandled exception hook (setUnhandledExceptionHook()), or make sure to call terminateWithUnhandledException() at the end of your hook. This will help you send exceptions to a third-party crash reporting service (like Firebase Crashlytics) and then terminate the program. Exceptions that escape main() and exceptions that cross the interop boundary will always terminate the program, even if the hook did not call terminateWithUnhandledException().

Kotlin/JS

We're continuing to work on stabilizing the IR backend for the Kotlin/JS compiler. Kotlin/JS now has an option to disable downloading of Node.js and Yarn.

Option to use pre-installed Node.js and Yarn

You can now disable downloading Node.js and Yarn when building Kotlin/JS projects and use the instances already installed on the host. This is useful for building on servers without internet connectivity, such as CI servers.

To disable downloading external components, add the following lines to your build.gradle(.kts):

- Yarn:

Kotlin

```kotlin
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().download = false // or true for default
behavior
}
```

Groovy

```groovy
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).download = false
}
```

- Node.js:

Kotlin

```kotlin
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension>().download = false // or true for default
behavior
}
```

Groovy

```groovy
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension).download = false
}
```

Kotlin Gradle plugin

In Kotlin 1.6.0, we changed the deprecation level of the KotlinGradleSubplugin class to 'ERROR'. This class was used for writing compiler plugins. In the following releases, we'll remove this class. Use the class KotlinCompilerPluginSupportPlugin instead.

We removed the kotlin.useFallbackCompilerSearch build option and the noReflect and includeRuntime compiler options. The useIR compiler option has been hidden and will be removed in upcoming releases.

Learn more about the currently supported compiler options in the Kotlin Gradle plugin.

Standard library

The new 1.6.0 version of the standard library stabilizes experimental features, introduces new ones, and unifies its behavior across the platforms:

- New readline functions
- Stable typeOf()
- Stable collection builders
- Stable Duration API
- Splitting Regex into a sequence
- Bit rotation operations on integers
- Changes for replace() and replaceFirst() in JS
- Improvements to the existing API
- Deprecations

New readline functions

Kotlin 1.6.0 offers new functions for handling standard input: readln() and readlnOrNull().

> For now, new functions are available for the JVM and Native target platforms only.

Earlier versions	1.6.0 alternative	Usage
readLine()!!	readln()	Reads a line from stdin and returns it, or throws a RuntimeException if EOF has been reached.
readLine()	readlnOrNull()	Reads a line from stdin and returns it, or returns null if EOF has been reached.

We believe that eliminating the need to use !! when reading a line will improve the experience for newcomers and simplify teaching Kotlin. To make the read-line operation name consistent with its println() counterpart, we've decided to shorten the names of new functions to 'ln'.

```
println("What is your nickname?")
val nickname = readln()
println("Hello, $nickname!")
```

```
fun main() {
//sampleStart
    var sum = 0
    while (true) {
        val nextLine = readlnOrNull().takeUnless {
            it.isNullOrEmpty()
        } ?: break
        sum += nextLine.toInt()
    }
    println(sum)
//sampleEnd
}
```

The existing readLine() function will get a lower priority than readln() and readlnOrNull() in your IDE code completion. IDE inspections will also recommend using new functions instead of the legacy readLine().

We're planning to gradually deprecate the readLine() function in future releases.

Stable typeOf()

Version 1.6.0 brings a Stable typeOf() function, closing one of the major roadmap items.

Since 1.3.40, typeOf() was available on the JVM platform as an experimental API. Now you can use it in any Kotlin platform and get KType representation of any Kotlin type that the compiler can infer:

```
inline fun <reified T> renderType(): String {
    val type = typeOf<T>()
    return type.toString()
}

fun main() {
    val fromExplicitType = typeOf<Int>()
    val fromReifiedType = renderType<List<Int>>()
}
```

Stable collection builders

In Kotlin 1.6.0, collection builder functions have been promoted to Stable. Collections returned by collection builders are now serializable in their read-only state.

You can now use buildMap(), buildList(), and buildSet() without the opt-in annotation:

```
fun main() {
//sampleStart
    val x = listOf('b', 'c')
    val y = buildList {
        add('a')
        addAll(x)
        add('d')
    }
    println(y)  // [a, b, c, d]
//sampleEnd
}
```

Stable Duration API

The Duration class for representing duration amounts in different time units has been promoted to Stable. In 1.6.0, the Duration API has received the following changes:

- The first component of the toComponents() function that decomposes the duration into days, hours, minutes, seconds, and nanoseconds now has the Long type instead of Int. Before, if the value didn't fit into the Int range, it was coerced into that range. With the Long type, you can decompose any value in the duration range without cutting off the values that don't fit into Int.

- The DurationUnit enum is now standalone and not a type alias of java.util.concurrent.TimeUnit on the JVM. We haven't found any convincing cases in which having typealias DurationUnit = TimeUnit could be useful. Also, exposing the TimeUnit API through a type alias might confuse DurationUnit users.
- In response to community feedback, we're bringing back extension properties like Int.seconds. But we'd like to limit their applicability, so we put them into the companion of the Duration class. While the IDE can still propose extensions in completion and automatically insert an import from the companion, in the future we plan to limit this behavior to cases when the Duration type is expected.

```
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
    val duration = 10000
    println("There are ${duration.seconds.inWholeMinutes} minutes in $duration seconds")
    // There are 166 minutes in 10000 seconds
//sampleEnd
}
```

We suggest replacing previously introduced companion functions, such as Duration.seconds(Int), and deprecated top-level extensions like Int.seconds with new extensions in Duration.Companion.

> Such a replacement may cause ambiguity between old top-level extensions and new companion extensions. Be sure to use the wildcard import of the kotlin.time package – import kotlin.time.* – before doing automated migration.

Splitting Regex into a sequence

The Regex.splitToSequence(CharSequence) and CharSequence.splitToSequence(Regex) functions are promoted to Stable. They split the string around matches of the given regex, but return the result as a Sequence so that all operations on this result are executed lazily:

```
fun main() {
//sampleStart
    val colorsText = "green, red, brown&blue, orange, pink&green"
    val regex = "[,\\s]+".toRegex()
    val mixedColor = regex.splitToSequence(colorsText)
    // or
    // val mixedColor = colorsText.splitToSequence(regex)
        .onEach { println(it) }
        .firstOrNull { it.contains('&') }
    println(mixedColor) // "brown&blue"
//sampleEnd
}
```

Bit rotation operations on integers

In Kotlin 1.6.0, the rotateLeft() and rotateRight() functions for bit manipulations became Stable. The functions rotate the binary representation of the number left or right by a specified number of bits:

```
fun main() {
//sampleStart
    val number: Short = 0b10001
    println(number
        .rotateRight(2)
        .toString(radix = 2)) // 100000000000100
    println(number
        .rotateLeft(2)
        .toString(radix = 2))  // 1000100
//sampleEnd
}
```

Changes for replace() and replaceFirst() in JS

Before Kotlin 1.6.0, the replace() and replaceFirst() Regex functions behaved differently in Java and JS when the replacement string contained a group reference. To make the behavior consistent across all target platforms, we've changed their implementation in JS.

Occurrences of ${name} or $index in the replacement string are substituted with the subsequences corresponding to the captured groups with the specified index or a name:

- $index – the first digit after '$' is always treated as a part of the group reference. Subsequent digits are incorporated into the index only if they form a valid group

reference.Only digits '0'–'9' are considered potential components of the group reference. Note that indexes of captured groups start from '1'. The group with index '0' stands for the whole match.

- ${name} – the name can consist of Latin letters 'a'–'z', 'A'–'Z', or digits '0'–'9'. The first character must be a letter.

> Named groups in replacement patterns are currently supported only on the JVM.

- To include the succeeding character as a literal in the replacement string, use the backslash character \:

```
fun main() {
//sampleStart
    println(Regex("(.+)").replace("Kotlin", """\$ $1""")) // $ Kotlin
    println(Regex("(.+)").replaceFirst("1.6.0", """\\ $1""")) // \ 1.6.0
//sampleEnd
}
```

You can use Regex.escapeReplacement() if the replacement string has to be treated as a literal string.

Improvements to the existing API

- Version 1.6.0 added the infix extension function for Comparable.compareTo(). You can now use the infix form for comparing two objects for order:

```
 class WrappedText(val text: String) : Comparable<WrappedText> {
     override fun compareTo(other: WrappedText): Int =
         this.text compareTo other.text
}
```

- Regex.replace() in JS is now also not inline to unify its implementation across all platforms.

- The compareTo() and equals() String functions, as well as the isBlank() CharSequence function now behave in JS exactly the same way they do on the JVM. Previously there were deviations when it came to non-ASCII characters.

Deprecations

In Kotlin 1.6.0, we're starting the deprecation cycle with a warning for some JS-only stdlib API.

concat(), match(), and matches() string functions

- To concatenate the string with the string representation of a given other object, use plus() instead of concat().

- To find all occurrences of a regular expression within the input, use findAll() of the Regex class instead of String.match(regex: String).

- To check if the regular expression matches the entire input, use matches() of the Regex class instead of String.matches(regex: String).

sort() on arrays taking comparison functions

We've deprecated the Array<out T>.sort() function and the inline functions ByteArray.sort(), ShortArray.sort(), IntArray.sort(), LongArray.sort(), FloatArray.sort(), DoubleArray.sort(), and CharArray.sort(), which sorted arrays following the order passed by the comparison function. Use other standard library functions for array sorting.

See the collection ordering section for reference.

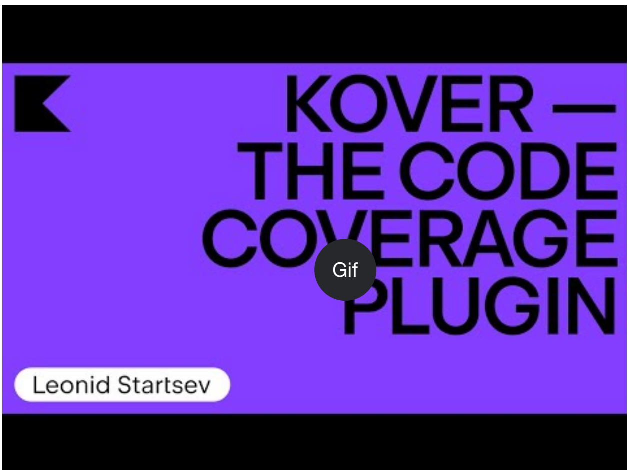
Tools

Kover – a code coverage tool for Kotlin

> The Kover Gradle plugin is Experimental. We would appreciate your feedback on it in GitHub.

With Kotlin 1.6.0, we're introducing Kover – a Gradle plugin for the IntelliJ and JaCoCo Kotlin code coverage agents. It works with all language constructs, including inline functions.

Learn more about Kover on its GitHub repository or in this video:



Watch video online.

Coroutines 1.6.0-RC

kotlinx.coroutines 1.6.0-RC is out with multiple features and improvements:

- Support for the new Kotlin/Native memory manager
- Introduction of dispatcher views API, which allows limiting parallelism without creating additional threads
- Migrating from Java 6 to Java 8 target
- kotlinx-coroutines-test with the new reworked API and multiplatform support
- Introduction of CopyableThreadContextElement, which gives coroutines a thread-safe write access to ThreadLocal variables

Learn more in the changelog.

Migrating to Kotlin 1.6.0

IntelliJ IDEA and Android Studio will suggest updating the Kotlin plugin to 1.6.0 once it is available.

To migrate existing projects to Kotlin 1.6.0, change the Kotlin version to 1.6.0 and reimport your Gradle or Maven project. Learn how to update to Kotlin 1.6.0.

To start a new project with Kotlin 1.6.0, update the Kotlin plugin and run the Project Wizard from File | New | Project.

The new command-line compiler is available for download on the GitHub release page.

Kotlin 1.6.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of such changes in the Compatibility Guide for Kotlin 1.6.

What's new in Kotlin 1.5.30

Released: 24 August 2021

Kotlin 1.5.30 offers language updates including previews of future changes, various improvements in platform support and tooling, and new standard library functions.

Here are some major improvements:

- Language features, including experimental sealed when statements, changes in using opt-in requirement, and others

- Native support for Apple silicon
- Kotlin/JS IR backend reaches Beta
- Improved Gradle plugin experience

You can also find a short overview of the changes in the release blog post and this video:

Watch video online.

Language features

Kotlin 1.5.30 is presenting previews of future language changes and bringing improvements to the opt-in requirement mechanism and type inference:

- Exhaustive when statements for sealed and Boolean subjects
- Suspending functions as supertypes
- Requiring opt-in on implicit usages of experimental APIs
- Changes to using opt-in requirement annotations with different targets
- Improvements to type inference for recursive generic types
- Eliminating builder inference restrictions

Exhaustive when statements for sealed and Boolean subjects

> Support for sealed (exhaustive) when statements is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

An exhaustive when statement contains branches for all possible types or values of its subject or for some types plus an else branch. In other words, it covers all possible cases.

We're planning to prohibit non-exhaustive when statements soon to make the behavior consistent with when expressions. To ensure smooth migration, you can configure the compiler to report warnings about non-exhaustive when statements with a sealed class or a Boolean. Such warnings will appear by default in Kotlin 1.6 and will become errors later.

> Enums already get a warning.

```
sealed class Mode {
    object ON : Mode()
```

```
    object OFF : Mode()
}

fun main() {
    val x: Mode = Mode.ON
    when (x) {
        Mode.ON -> println("ON")
    }
// WARNING: Non exhaustive 'when' statements on sealed classes/interfaces
// will be prohibited in 1.7, add an 'OFF' or 'else' branch instead

    val y: Boolean = true
    when (y) {
        true -> println("true")
    }
// WARNING: Non exhaustive 'when' statements on Booleans will be prohibited
// in 1.7, add a 'false' or 'else' branch instead
}
```

To enable this feature in Kotlin 1.5.30, use language version 1.6. You can also change the warnings to errors by enabling progressive mode.

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.6"
            //progressiveMode = true // false by default
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.6'
            //progressiveMode = true // false by default
        }
    }
}
```

Suspending functions as supertypes

> Support for suspending functions as supertypes is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.5.30 provides a preview of the ability to use a suspend functional type as a supertype with some limitations.

```
class MyClass: suspend () -> Unit {
    override suspend fun invoke() { TODO() }
}
```

Use the -language-version 1.6 compiler option to enable the feature:

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.6"
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.6'
        }
    }
}
```

The feature has the following restrictions:

- You can't mix an ordinary functional type and a suspend functional type as supertype. This is because of the implementation details of suspend functional types in the JVM backend. They are represented in it as ordinary functional types with a marker interface. Because of the marker interface, there is no way to tell which of the superinterfaces are suspended and which are ordinary.
- You can't use multiple suspend functional supertypes. If there are type checks, you also can't use multiple ordinary functional supertypes.

Requiring opt-in on implicit usages of experimental APIs

> The opt-in requirement mechanism is Experimental. It may change at any time. See how to opt-in. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The author of a library can mark an experimental API as requiring opt-in to inform users about its experimental state. The compiler raises a warning or error when the API is used and requires explicit consent to suppress it.

In Kotlin 1.5.30, the compiler treats any declaration that has an experimental type in the signature as experimental. Namely, it requires opt-in even for implicit usages of an experimental API. For example, if the function's return type is marked as an experimental API element, a usage of the function requires you to opt-in even if the declaration is not marked as requiring an opt-in explicitly.

```kotlin
// Library code

@RequiresOptIn(message = "This API is experimental.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS)
annotation class MyDateTime // Opt-in requirement annotation

@MyDateTime
class DateProvider // A class requiring opt-in

// Client code

// Warning: experimental API usage
fun createDateSource(): DateProvider { /* ... */ }

fun getDate(): Date {
    val dateSource = createDateSource() // Also warning: experimental API usage
    // ...
}
```

Learn more about opt-in requirements.

Changes to using opt-in requirement annotations with different targets

> The opt-in requirement mechanism is Experimental. It may change at any time. See how to opt-in. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.5.30 presents new rules for using and declaring opt-in requirement annotations on different targets. The compiler now reports an error for use cases that are impractical to handle at compile time. In Kotlin 1.5.30:

- Marking local variables and value parameters with opt-in requirement annotations is forbidden at the use site.

- Marking override is allowed only if its basic declaration is also marked.
- Marking backing fields and getters is forbidden. You can mark the basic property instead.
- Setting TYPE and TYPE_PARAMETER annotation targets is forbidden at the opt-in requirement annotation declaration site.

Learn more about opt-in requirements.

Improvements to type inference for recursive generic types

In Kotlin and Java, you can define a recursive generic type, which references itself in its type parameters. In Kotlin 1.5.30, the Kotlin compiler can infer a type argument based only on upper bounds of the corresponding type parameter if it is a recursive generic. This makes it possible to create various patterns with recursive generic types that are often used in Java to make builder APIs.

```
// Kotlin 1.5.20
val containerA = PostgreSQLContainer<Nothing>(DockerImageName.parse("postgres:13-alpine")).apply {
    withDatabaseName("db")
    withUsername("user")
    withPassword("password")
    withInitScript("sql/schema.sql")
}

// Kotlin 1.5.30
val containerB = PostgreSQLContainer(DockerImageName.parse("postgres:13-alpine"))
    .withDatabaseName("db")
    .withUsername("user")
    .withPassword("password")
    .withInitScript("sql/schema.sql")
```

You can enable the improvements by passing the -Xself-upper-bound-inference or the -language-version 1.6 compiler options. See other examples of newly supported use cases in this YouTrack ticket.

Eliminating builder inference restrictions

Builder inference is a special kind of type inference that allows you to infer the type arguments of a call based on type information from other calls inside its lambda argument. This can be useful when calling generic builder functions such as buildList() or sequence(): buildList { add("string") }.

Inside such a lambda argument, there was previously a limitation on using the type information that the builder inference tries to infer. This means you can only specify it and cannot get it. For example, you cannot call get() inside a lambda argument of buildList() without explicitly specified type arguments.

Kotlin 1.5.30 removes these limitations with the -Xunrestricted-builder-inference compiler option. Add this option to enable previously prohibited calls inside a lambda argument of generic builder functions:

```
@kotlin.ExperimentalStdlibApi
val list = buildList {
    add("a")
    add("b")
    set(1, null)
    val x = get(1)
    if (x != null) {
        removeAt(1)
    }
}

@kotlin.ExperimentalStdlibApi
val map = buildMap {
    put("a", 1)
    put("b", 1.1)
    put("c", 2f)
}
```

Also, you can enable this feature with the -language-version 1.6 compiler option.

Kotlin/JVM

With Kotlin 1.5.30, Kotlin/JVM receives the following features:

- Instantiation of annotation classes

* Improved nullability annotation support configuration

See the Gradle section for Kotlin Gradle plugin updates on the JVM platform.

Instantiation of annotation classes

> Instantiation of annotation classes is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

With Kotlin 1.5.30 you can now call constructors of annotation classes in arbitrary code to obtain a resulting instance. This feature covers the same use cases as the Java convention that allows the implementation of an annotation interface.

```
annotation class InfoMarker(val info: String)

fun processInfo(marker: InfoMarker) = ...

fun main(args: Array<String>) {
    if (args.size != 0)
        processInfo(getAnnotationReflective(args))
    else
        processInfo(InfoMarker("default"))
}
```

Use the -language-version 1.6 compiler option to enable this feature. Note that all current annotation class limitations, such as restrictions to define non-val parameters or members different from secondary constructors, remain intact.

Learn more about instantiation of annotation classes in this KEEP

Improved nullability annotation support configuration

The Kotlin compiler can read various types of nullability annotations to get nullability information from Java. This information allows it to report nullability mismatches in Kotlin when calling Java code.

In Kotlin 1.5.30, you can specify whether the compiler reports a nullability mismatch based on the information from specific types of nullability annotations. Just use the compiler option -Xnullability-annotations=@<package-name>:<report-level>. In the argument, specify the fully qualified nullability annotations package and one of these report levels:

* ignore to ignore nullability mismatches
* warn to report warnings
* strict to report errors.

See the full list of supported nullability annotations along with their fully qualified package names.

Here is an example showing how to enable error reporting for the newly supported RxJava 3 nullability annotations: -Xnullability-annotations=@io.reactivex.rxjava3.annotations:strict. Note that all such nullability mismatches are warnings by default.

Kotlin/Native

Kotlin/Native has received various changes and improvements:

* Apple silicon support
* Improved Kotlin DSL for the CocoaPods Gradle plugin
* Experimental interoperability with Swift 5.5 async/await
* Improved Swift/Objective-C mapping for objects and companion objects
* Deprecation of linkage against DLLs without import libraries for MinGW targets

Apple silicon support

Kotlin 1.5.30 introduces native support for Apple silicon.

Previously, the Kotlin/Native compiler and tooling required the Rosetta translation environment for working on Apple silicon hosts. In Kotlin 1.5.30, the translation environment is no longer needed – the compiler and tooling can run on Apple silicon hardware without requiring any additional actions.

We've also introduced new targets that make Kotlin code run natively on Apple silicon:

- macosArm64
- iosSimulatorArm64
- watchosSimulatorArm64
- tvosSimulatorArm64

They are available on both Intel-based and Apple silicon hosts. All existing targets are available on Apple silicon hosts as well.

Note that in 1.5.30 we provide only basic support for Apple silicon targets in the kotlin-multiplatform Gradle plugin. Particularly, the new simulator targets aren't included in the ios, tvos, and watchos target shortcuts. Learn how to use Apple silicon targets with the target shortcuts. We will keep working to improve the user experience with the new targets.

Improved Kotlin DSL for the CocoaPods Gradle plugin

New parameters for Kotlin/Native frameworks

Kotlin 1.5.30 introduces the improved CocoaPods Gradle plugin DSL for Kotlin/Native frameworks. In addition to the name of the framework, you can specify other parameters in the pod configuration:

- Specify the dynamic or static version of the framework
- Enable export dependencies explicitly
- Enable Bitcode embedding

To use the new DSL, update your project to Kotlin 1.5.30, and specify the parameters in the cocoapods section of your build.gradle(.kts) file:

```
cocoapods {
    frameworkName = "MyFramework" // This property is deprecated
    // and will be removed in future versions
    // New DSL for framework configuration:
    framework {
        // All Framework properties are supported
        // Framework name configuration. Use this property instead of
        // deprecated 'frameworkName'
        baseName = "MyFramework"
        // Dynamic framework support
        isStatic = false
        // Dependency export
        export(project(":anotherKMMModule"))
        transitiveExport = false // This is default.
        // Bitcode embedding
        embedBitcode(BITCODE)
    }
}
```

Support custom names for Xcode configuration

The Kotlin CocoaPods Gradle plugin supports custom names in the Xcode build configuration. It will also help you if you're using special names for the build configuration in Xcode, for example Staging.

To specify a custom name, use the xcodeConfigurationToNativeBuildType parameter in the cocoapods section of your build.gradle(.kts) file:

```
cocoapods {
    // Maps custom Xcode configuration to NativeBuildType
    xcodeConfigurationToNativeBuildType["CUSTOM_DEBUG"] = NativeBuildType.DEBUG
    xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] = NativeBuildType.RELEASE
}
```

This parameter will not appear in the Podspec file. When Xcode runs the Gradle build process, the Kotlin CocoaPods Gradle plugin will select the necessary native build type.

There's no need to declare the Debug and Release configurations because they are supported by default.

Experimental interoperability with Swift 5.5 async/await

Concurrency interoperability with Swift async/await is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

We added support for calling Kotlin's suspending functions from Objective-C and Swift in 1.4.0, and now we're improving it to keep up with a new Swift 5.5 feature – concurrency with async and await modifiers.

The Kotlin/Native compiler now emits the _Nullable_result attribute in the generated Objective-C headers for suspending functions with nullable return types. This makes it possible to call them from Swift as async functions with the proper nullability.

Note that this feature is experimental and can be affected in the future by changes in both Kotlin and Swift. For now, we're offering a preview of this feature that has certain limitations, and we are eager to hear what you think. Learn more about its current state and leave your feedback in this YouTrack issue.

Improved Swift/Objective-C mapping for objects and companion objects

Getting objects and companion objects can now be done in a way that is more intuitive for native iOS developers. For example, if you have the following objects in Kotlin:

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

To access them in Swift, you can use the shared and companion properties:

```
MyObject.shared
MyObject.shared.x
MyClass.companion
MyClass.Companion.shared
```

Learn more about Swift/Objective-C interoperability.

Deprecation of linkage against DLLs without import libraries for MinGW targets

LLD is a linker from the LLVM project, which we plan to start using in Kotlin/Native for MinGW targets because of its benefits over the default ld.bfd – primarily its better performance.

However, the latest stable version of LLD doesn't support direct linkage against DLL for MinGW (Windows) targets. Such linkage requires using import libraries. Although they aren't needed with Kotlin/Native 1.5.30, we're adding a warning to inform you that such usage is incompatible with LLD that will become the default linker for MinGW in the future.

Please share your thoughts and concerns about the transition to the LLD linker in this YouTrack issue.

Kotlin Multiplatform

1.5.30 brings the following notable updates to Kotlin Multiplatform:

- Ability to use custom cinterop libraries in shared native code
- Support for XCFrameworks
- New default publishing setup for Android artifacts

Ability to use custom cinterop libraries in shared native code

Kotlin Multiplatform gives you an option to use platform-dependent interop libraries in shared source sets. Before 1.5.30, this worked only with platform libraries shipped with Kotlin/Native distribution. Starting from 1.5.30, you can use it with your custom cinterop libraries. To enable this feature, add the kotlin.mpp.enableCInteropCommonization=true property in your gradle.properties:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true
kotlin.native.enableDependencyPropagation=false
kotlin.mpp.enableCInteropCommonization=true
```

Support for XCFrameworks

All Kotlin Multiplatform projects can now have XCFrameworks as an output format. Apple introduced XCFrameworks as a replacement for universal (fat) frameworks. With the help of XCFrameworks you:

- Can gather logic for all the target platforms and architectures in a single bundle.
- Don't need to remove all unnecessary architectures before publishing the application to the App Store.

XCFrameworks is useful if you want to use your Kotlin framework for devices and simulators on Apple M1.

To use XCFrameworks, update your build.gradle(.kts) script:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFramework

plugins {
    kotlin("multiplatform")
}

kotlin {
    val xcf = XCFramework()

    ios {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
    watchos {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
    tvos {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFrameworkConfig

plugins {
    id 'org.jetbrains.kotlin.multiplatform'
}

kotlin {
    def xcf = new XCFrameworkConfig(project)

    ios {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
    watchos {
        binaries.framework {
            baseName = "shared"
```

```
            xcf.add(it)
        }
    }
    tvos {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
}
```

When you declare XCFrameworks, these new Gradle tasks will be registered:

- assembleXCFramework
- assembleDebugXCFramework (additionally debug artifact that contains dSYMs)
- assembleReleaseXCFramework

Learn more about XCFrameworks in this WWDC video.

New default publishing setup for Android artifacts

Using the maven-publish Gradle plugin, you can publish your multiplatform library for the Android target by specifying Android variant names in the build script. The Kotlin Gradle plugin will generate publications automatically.

Before 1.5.30, the generated publication metadata included the build type attributes for every published Android variant, making it compatible only with the same build type used by the library consumer. Kotlin 1.5.30 introduces a new default publishing setup:

- If all Android variants that the project publishes have the same build type attribute, then the published variants won't have the build type attribute and will be compatible with any build type.
- If the published variants have different build type attributes, then only those with the release value will be published without the build type attribute. This makes the release variants compatible with any build type on the consumer side, while non-release variants will only be compatible with the matching consumer build types.

To opt-out and keep the build type attributes for all variants, you can set this Gradle property: kotlin.android.buildTypeAttribute.keep=true.

Kotlin/JS

Two major improvements are coming to Kotlin/JS with 1.5.30:

- JS IR compiler backend reaches Beta
- Better debugging experience for applications with the Kotlin/JS IR backend

JS IR compiler backend reaches Beta

The IR-based compiler backend for Kotlin/JS, which was introduced in 1.4.0 in Alpha, has reached Beta.

Previously, we published the migration guide for the JS IR backend to help you migrate your projects to the new backend. Now we would like to present the Kotlin/JS Inspection Pack IDE plugin, which displays the required changes directly in IntelliJ IDEA.

Better debugging experience for applications with the Kotlin/JS IR backend

Kotlin 1.5.30 brings JavaScript source map generation for the Kotlin/JS IR backend. This will improve the Kotlin/JS debugging experience when the IR backend is enabled, with full debugging support that includes breakpoints, stepping, and readable stack traces with proper source references.

Learn how to debug Kotlin/JS in the browser or IntelliJ IDEA Ultimate.

Gradle

As a part of our mission to improve the Kotlin Gradle plugin user experience, we've implemented the following features:

- Support for Java toolchains, which includes an ability to specify a JDK home with the UsesKotlinJavaToolchain interface for older Gradle versions

- An easier way to explicitly specify the Kotlin daemon's JVM arguments

Support for Java toolchains

Gradle 6.7 introduced the "Java toolchains support" feature. Using this feature, you can:

- Run compilations, tests, and executables using JDKs and JREs that are different from the Gradle ones.
- Compile and test code with an unreleased language version.

With toolchains support, Gradle can autodetect local JDKs and install missing JDKs that Gradle requires for the build. Now Gradle itself can run on any JDK and still reuse the build cache feature.

The Kotlin Gradle plugin supports Java toolchains for Kotlin/JVM compilation tasks. A Java toolchain:

- Sets the jdkHome option available for JVM targets.

  > The ability to set the jdkHome option directly has been deprecated.

- Sets the kotlinOptions.jvmTarget to the toolchain's JDK version if the user didn't set the jvmTarget option explicitly. If the toolchain is not configured, the jvmTarget field uses the default value. Learn more about JVM target compatibility.
- Affects which JDK kapt workers are running on.

Use the following code to set a toolchain. Replace the placeholder <MAJOR_JDK_VERSION> with the JDK version you would like to use:

Kotlin

```kotlin
kotlin {
    jvmToolchain {
        (this as JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
    }
}
```

Groovy

```groovy
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
    }
}
```

Note that setting a toolchain via the kotlin extension will update the toolchain for Java compile tasks as well.

You can set a toolchain via the java extension, and Kotlin compilation tasks will use it:

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
    }
}
```

For information about setting any JDK version for KotlinCompile tasks, look through the docs about setting the JDK version with the Task DSL.

For Gradle versions from 6.1 to 6.6, use the UsesKotlinJavaToolchain interface to set the JDK home.

Ability to specify JDK home with UsesKotlinJavaToolchain interface

All Kotlin tasks that support setting the JDK via kotlinOptions now implement the UsesKotlinJavaToolchain interface. To set the JDK home, put a path to your JDK and replace the <JDK_VERSION> placeholder:

Kotlin

```
project.tasks
    .withType<UsesKotlinJavaToolchain>()
    .configureEach {
        it.kotlinJavaToolchain.jdk.use(
            "/path/to/local/jdk",
            JavaVersion.<LOCAL_JDK_VERSION>
        )
    }
```

Groovy

```
project.tasks
    .withType(UsesKotlinJavaToolchain.class)
    .configureEach {
        it.kotlinJavaToolchain.jdk.use(
            '/path/to/local/jdk',
            JavaVersion.<LOCAL_JDK_VERSION>
        )
    }
```

Use the UsesKotlinJavaToolchain interface for Gradle versions from 6.1 to 6.6. Starting from Gradle 6.7, use the Java toolchains instead.

When using this feature, note that kapt task workers will only use process isolation mode, and the kapt.workers.isolation property will be ignored.

Easier way to explicitly specify Kotlin daemon JVM arguments

In Kotlin 1.5.30, there's a new logic for the Kotlin daemon's JVM arguments. Each of the options in the following list overrides the ones that came before it:

- If nothing is specified, the Kotlin daemon inherits arguments from the Gradle daemon (as before). For example, in the gradle.properties file:

```
org.gradle.jvmargs=-Xmx1500m -Xms=500m
```

- If the Gradle daemon's JVM arguments have the kotlin.daemon.jvm.options system property, use it as before:

```
org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m -Xms=500m
```

- You can add thekotlin.daemon.jvmargs property in the gradle.properties file:

```
kotlin.daemon.jvmargs=-Xmx1500m -Xms=500m
```

- You can specify arguments in the kotlin extension:

Kotlin

```
kotlin {
    kotlinDaemonJvmArgs = listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC")
}
```

Groovy

```
kotlin {
    kotlinDaemonJvmArgs = ["-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"]
}
```

- You can specify arguments for a specific task:

Kotlin

```
tasks
    .matching { it.name == "compileKotlin" && it is CompileUsingKotlinDaemon }
    .configureEach {
        (this as CompileUsingKotlinDaemon).kotlinDaemonJvmArguments.set(listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"))
```

```
    }
```

Groovy

```
tasks
    .matching {
        it.name == "compileKotlin" && it instanceof CompileUsingKotlinDaemon
    }
    .configureEach {
        kotlinDaemonJvmArguments.set(["-Xmx1g", "-Xms512m"])
    }
```

> In this case a new Kotlin daemon instance can start on task execution. Learn more about the Kotlin daemon's interactions with JVM arguments.

For more information about the Kotlin daemon, see the Kotlin daemon and using it with Gradle.

Standard library

Kotlin 1.5.30 is bringing improvements to the standard library's Duration and Regex APIs:

- Changing Duration.toString() output
- Parsing Duration from String
- Matching with Regex at a particular position
- Splitting Regex to a sequence

Changing Duration.toString() output

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.

Before Kotlin 1.5.30, the Duration.toString() function would return a string representation of its argument expressed in the unit that yielded the most compact and readable number value. From now on, it will return a string value expressed as a combination of numeric components, each in its own unit. Each component is a number followed by the unit's abbreviated name: d, h, m, s. For example:

Example of function call	Previous output	Current output
Duration.days(45).toString()	45.0d	45d
Duration.days(1.5).toString()	36.0h	1d 12h
Duration.minutes(1230).toString()	20.5h	20h 30m
Duration.minutes(2415).toString()	40.3h	1d 16h 15m
Duration.minutes(920).toString()	920m	15h 20m
Duration.seconds(1.546).toString()	1.55s	1.546s

Example of function call	Previous output	Current output
Duration.milliseconds(25.12).toString()	25.1ms	25.12ms

The way negative durations are represented has also been changed. A negative duration is prefixed with a minus sign (-), and if it consists of multiple components, it is surrounded with parentheses: -12m and -(1h 30m).

Note that small durations of less than one second are represented as a single number with one of the subsecond units. For example, ms (milliseconds), us (microseconds), or ns (nanoseconds): 140.884ms, 500us, 24ns. Scientific notation is no longer used to represent them.

If you want to express duration in a single unit, use the overloaded Duration.toString(unit, decimals) function.

> We recommend using Duration.toIsoString() in certain cases, including serialization and interchange. Duration.toIsoString() uses the stricter ISO-8601 format instead of Duration.toString().

Parsing Duration from String

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in this issue.

In Kotlin 1.5.30, there are new functions in the Duration API:

- parse(), which supports parsing the outputs of:
  - toString().
  - toString(unit, decimals).
  - toIsoString().
- parseIsoString(), which only parses from the format produced by toIsoString().
- parseOrNull() and parseIsoStringOrNull(), which behave like the functions above but return null instead of throwing IllegalArgumentException on invalid duration formats.

Here are some examples of parse() and parseOrNull() usages:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
//sampleStart
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    val singleUnitFormatString = "1.5h"
    val invalidFormatString = "1 hour 30 minutes"
    println(Duration.parse(isoFormatString)) // "1h 30m"
    println(Duration.parse(defaultFormatString)) // "1h 30m"
    println(Duration.parse(singleUnitFormatString)) // "1h 30m"
    //println(Duration.parse(invalidFormatString)) // throws exception
    println(Duration.parseOrNull(invalidFormatString)) // "null"
//sampleEnd
}
```

And here are some examples of parseIsoString() and parseIsoStringOrNull() usages:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
//sampleStart
```

```
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    println(Duration.parseIsoString(isoFormatString)) // "1h 30m"
    //println(Duration.parseIsoString(defaultFormatString)) // throws exception
    println(Duration.parseIsoStringOrNull(defaultFormatString)) // "null"
//sampleEnd
}
```

Matching with Regex at a particular position

Regex.matchAt() and Regex.matchesAt() functions are Experimental. They may be dropped or changed at any time. Use them only for evaluation purposes. We would appreciate hearing your feedback on them in YouTrack.

The new Regex.matchAt() and Regex.matchesAt() functions provide a way to check whether a regex has an exact match at a particular position in a String or CharSequence.

matchesAt() returns a boolean result:

```
fun main(){
//sampleStart
    val releaseText = "Kotlin 1.5.30 is released!"
    // regular expression: one digit, dot, one digit, dot, one or more digits
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()
    println(versionRegex.matchesAt(releaseText, 0)) // "false"
    println(versionRegex.matchesAt(releaseText, 7)) // "true"
//sampleEnd
}
```

matchAt() returns the match if one is found or null if one isn't:

```
fun main(){
//sampleStart
    val releaseText = "Kotlin 1.5.30 is released!"
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()
    println(versionRegex.matchAt(releaseText, 0)) // "null"
    println(versionRegex.matchAt(releaseText, 7)?.value) // "1.5.30"
//sampleEnd
}
```

Splitting Regex to a sequence

Regex.splitToSequence() and CharSequence.splitToSequence(Regex) functions are Experimental. They may be dropped or changed at any time. Use them only for evaluation purposes. We would appreciate hearing your feedback on them in YouTrack.

The new Regex.splitToSequence() function is a lazy counterpart of split(). It splits the string around matches of the given regex, but it returns the result as a Sequence so that all operations on this result are executed lazily.

```
fun main(){
//sampleStart
    val colorsText = "green, red , brown&blue, orange, pink&green"
    val regex = "[,\\s]+".toRegex()
    val mixedColor = regex.splitToSequence(colorsText)
        .onEach { println(it) }
        .firstOrNull { it.contains('&') }
    println(mixedColor) // "brown&blue"
//sampleEnd
}
```

A similar function was also added to CharSequence:

```
    val mixedColor = colorsText.splitToSequence(regex)
```

Serialization 1.3.0-RC

kotlinx.serialization 1.3.0-RC is here with new JSON serialization capabilities:

- Java IO streams serialization
- Property-level control over default values
- An option to exclude null values from serialization
- Custom class discriminators in polymorphic serialization

Learn more in the changelog.

What's new in Kotlin 1.5.20

Released: 24 June 2021

Kotlin 1.5.20 has fixes for issues discovered in the new features of 1.5.0, and it also includes various tooling improvements.

You can find an overview of the changes in the release blog post and this video:

Watch video online.

Kotlin/JVM

Kotlin 1.5.20 is receiving the following updates on the JVM platform:

- String concatenation via invokedynamic
- Support for JSpecify nullness annotations
- Support for calling Java's Lombok-generated methods within modules that have Kotlin and Java code

String concatenation via invokedynamic

Kotlin 1.5.20 compiles string concatenations into dynamic invocations (invokedynamic) on JVM 9+ targets, thereby keeping up with modern Java versions. More precisely, it uses StringConcatFactory.makeConcatWithConstants() for string concatenation.

To switch back to concatenation via StringBuilder.append() used in previous versions, add the compiler option -Xstring-concat=inline.

Learn how to add compiler options in Gradle, Maven, and the command-line compiler.

Support for JSpecify nullness annotations

The Kotlin compiler can read various types of nullability annotations to pass nullability information from Java to Kotlin. Version 1.5.20 introduces support for the

JSpecify project, which includes the standard unified set of Java nullness annotations.

With JSpecify, you can provide more detailed nullability information to help Kotlin keep null-safety interoperating with Java. You can set default nullability for the declaration, package, or module scope, specify parametric nullability, and more. You can find more details about this in the JSpecify user guide.

Here is the example of how Kotlin can handle JSpecify annotations:

```
// JavaClass.java
import org.jspecify.nullness.*;

@NullMarked
public class JavaClass {
  public String notNullableString() { return ""; }
  public @Nullable String nullableString() { return ""; }
}
```

```
// Test.kt
fun kotlinFun() = with(JavaClass()) {
  notNullableString().length // OK
  nullableString().length    // Warning: receiver nullability mismatch
}
```

In 1.5.20, all nullability mismatches according to the JSpecify-provided nullability information are reported as warnings. Use the -Xjspecify-annotations=strict and -Xtype-enhancement-improvements-strict-mode compiler options to enable strict mode (with error reporting) when working with JSpecify. Please note that the JSpecify project is under active development. Its API and implementation can change significantly at any time.

Learn more about null-safety and platform types.

Support for calling Java's Lombok-generated methods within modules that have Kotlin and Java code

> The Lombok compiler plugin is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.5.20 introduces an experimental Lombok compiler plugin. This plugin makes it possible to generate and use Java's Lombok declarations within modules that have Kotlin and Java code. Lombok annotations work only in Java sources and are ignored if you use them in Kotlin code.

The plugin supports the following annotations:

- @Getter, @Setter
- @NoArgsConstructor, @RequiredArgsConstructor, and @AllArgsConstructor
- @Data
- @With
- @Value

We're continuing to work on this plugin. To find out the detailed current state, visit the Lombok compiler plugin's README.

Currently, we don't have plans to support the @Builder annotation. However, we can consider this if you vote for @Builder in YouTrack.

Learn how to configure the Lombok compiler plugin.

Kotlin/Native

Kotlin/Native 1.5.20 offers a preview of the new feature and the tooling improvements:

- Opt-in export of KDoc comments to generated Objective-C headers
- Compiler bug fixes
- Improved performance of Array.copyInto() inside one array

Opt-in export of KDoc comments to generated Objective-C headers

> The ability to export KDoc comments to generated Objective-C headers is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

You can now set the Kotlin/Native compiler to export the documentation comments (KDoc) from Kotlin code to the Objective-C frameworks generated from it, making them visible to the frameworks' consumers.

For example, the following Kotlin code with KDoc:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
fun printSum(a: Int, b: Int) = println(a.toLong() + b)
```

produces the following Objective-C headers:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
+ (void)printSumA:(int32_t)a b:(int32_t)b __attribute__((swift_name("printSum(a:b:)")));
```

This also works well with Swift.

To try out this ability to export KDoc comments to Objective-C headers, use the -Xexport-kdoc compiler option. Add the following lines to build.gradle(.kts) of the Gradle projects you want to export comments from:

Kotlin

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        compilations.get("main").kotlinOptions.freeCompilerArgs += "-Xexport-kdoc"
    }
}
```

Groovy

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        compilations.get("main").kotlinOptions.freeCompilerArgs += "-Xexport-kdoc"
    }
}
```

We would be very grateful if you would share your feedback with us using this YouTrack ticket.

Compiler bug fixes

The Kotlin/Native compiler has received multiple bug fixes in 1.5.20. You can find the complete list in the changelog.

There is an important bug fix that affects compatibility: in previous versions, string constants that contained incorrect UTF surrogate pairs were losing their values during compilation. Now such values are preserved. Application developers can safely update to 1.5.20 – nothing will break. However, libraries compiled with 1.5.20 are incompatible with earlier compiler versions. See this YouTrack issue for details.

Improved performance of Array.copyInto() inside one array

We've improved the way Array.copyInto() works when its source and destination are the same array. Now such operations finish up to 20 times faster (depending on the number of objects being copied) due to memory management optimizations for this use case.

Kotlin/JS

With 1.5.20, we're publishing a guide that will help you migrate your projects to the new IR-based backend for Kotlin/JS.

Migration guide for the JS IR backend

The new migration guide for the JS IR backend identifies issues you may encounter during migration and provides solutions for them. If you find any issues that aren't covered in the guide, please report them to our issue tracker.

Gradle

Kotlin 1.5.20 introduces the following features that can improve the Gradle experience:

- Caching for annotation processors classloaders in kapt
- Deprecation of the kotlin.parallel.tasks.in.project build property

Caching for annotation processors' classloaders in kapt

> Caching for annotation processors' classloaders in kapt is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

There is now a new experimental feature that makes it possible to cache the classloaders of annotation processors in kapt. This feature can increase the speed of kapt for consecutive Gradle runs.

To enable this feature, use the following properties in your gradle.properties file:

```
# positive value will enable caching
# use the same value as the number of modules that use kapt
kapt.classloaders.cache.size=5

# disable for caching to work
kapt.include.compile.classpath=false
```

Learn more about kapt.

Deprecation of the kotlin.parallel.tasks.in.project build property

With this release, Kotlin parallel compilation is controlled by the Gradle parallel execution flag --parallel. Using this flag, Gradle executes tasks concurrently, increasing the speed of compiling tasks and utilizing the resources more efficiently.

You no longer need to use the kotlin.parallel.tasks.in.project property. This property has been deprecated and will be removed in the next major release.

Standard library

Kotlin 1.5.20 changes the platform-specific implementations of several functions for working with characters and as a result brings unification across platforms:

- Support for all Unicode digits in Char.digitToInt() for Kotlin/Native and Kotlin/JS.
- Unification of Char.isLowerCase()/isUpperCase() implementations across platforms.

Support for all Unicode digits in Char.digitToInt() in Kotlin/Native and Kotlin/JS

Char.digitToInt() returns the numeric value of the decimal digit that the character represents. Before 1.5.20, the function supported all Unicode digit characters only for Kotlin/JVM: implementations on the Native and JS platforms supported only ASCII digits.

From now, both with Kotlin/Native and Kotlin/JS, you can call Char.digitToInt() on any Unicode digit character and get its numeric representation.

```kotlin
fun main() {
//sampleStart
    val ten = '\u0661'.digitToInt() + '\u0039'.digitToInt() // ARABIC-INDIC DIGIT ONE + DIGIT NINE
    println(ten)
//sampleEnd
```

```
}
```

Unification of Char.isLowerCase()/isUpperCase() implementations across platforms

The functions Char.isUpperCase() and Char.isLowerCase() return a boolean value depending on the case of the character. For Kotlin/JVM, the implementation checks both the General_Category and the Other_Uppercase/Other_Lowercase Unicode properties.

Prior to 1.5.20, implementations for other platforms worked differently and considered only the general category. In 1.5.20, implementations are unified across platforms and use both properties to determine the character case:

```
fun main() {
//sampleStart
    val latinCapitalA = 'A' // has "Lu" general category
    val circledLatinCapitalA = 'Ⓐ' // has "Other_Uppercase" property
    println(latinCapitalA.isUpperCase() && circledLatinCapitalA.isUpperCase())
//sampleEnd
}
```

What's new in Kotlin 1.5.0

Released: 5 May 2021

Kotlin 1.5.0 introduces new language features, stable IR-based JVM compiler backend, performance improvements, and evolutionary changes such as stabilizing experimental features and deprecating outdated ones.

You can also find an overview of the changes in the release blog post.

Language features

Kotlin 1.5.0 brings stable versions of the new language features presented for preview in 1.4.30:

- JVM records support
- Sealed interfaces and sealed class improvements
- Inline classes

Detailed descriptions of these features are available in this blog post and the corresponding pages of Kotlin documentation.

JVM records support

Java is evolving fast, and to make sure Kotlin remains interoperable with it, we've introduced support for one of its latest features – record classes.

Kotlin's support for JVM records includes bidirectional interoperability:

- In Kotlin code, you can use Java record classes like you would use typical classes with properties.
- To use a Kotlin class as a record in Java code, make it a data class and mark it with the @JvmRecord annotation.

```
@JvmRecord
data class User(val name: String, val age: Int)
```

Learn more about using JVM records in Kotlin.

Watch video online.

Sealed interfaces

Kotlin interfaces can now have the sealed modifier, which works on interfaces in the same way it works on classes: all implementations of a sealed interface are known at compile time.

```
sealed interface Polygon
```

You can rely on that fact, for example, to write exhaustive when expressions.

```
fun draw(polygon: Polygon) = when (polygon) {
    is Rectangle -> // ...
    is Triangle -> // ...
    // else is not needed - all possible implementations are covered
}
```

Additionally, sealed interfaces enable more flexible restricted class hierarchies because a class can directly inherit more than one sealed interface.

```
class FilledRectangle: Polygon, Fillable
```

Learn more about sealed interfaces.

Watch video online.

Package-wide sealed class hierarchies

Sealed classes can now have subclasses in all files of the same compilation unit and the same package. Previously, all subclasses had to appear in the same file.

Direct subclasses may be top-level or nested inside any number of other named classes, named interfaces, or named objects.

The subclasses of a sealed class must have a name that is properly qualified – they cannot be local or anonymous objects.

Learn more about sealed class hierarchies.

Inline classes

Inline classes are a subset of value-based classes that only hold values. You can use them as wrappers for a value of a certain type without the additional overhead that comes from using memory allocations.

Inline classes can be declared with the value modifier before the name of the class:

```
value class Password(val s: String)
```

The JVM backend also requires a special @JvmInline annotation:

```
@JvmInline
value class Password(val s: String)
```

The inline modifier is now deprecated with a warning.

Learn more about inline classes.

Watch video online.

Kotlin/JVM

Kotlin/JVM has received a number of improvements, both internal and user-facing. Here are the most notable among them:

- Stable JVM IR backend
- New default JVM target: 1.8
- SAM adapters via invokedynamic
- Lambdas via invokedynamic
- Deprecation of @JvmDefault and old Xjvm-default modes
- Improvements to handling nullability annotations

Stable JVM IR backend

The IR-based backend for the Kotlin/JVM compiler is now Stable and enabled by default.

Starting from Kotlin 1.4.0, early versions of the IR-based backend were available for preview, and it has now become the default for language version 1.5. The old backend is still used by default for earlier language versions.

You can find more details about the benefits of the IR backend and its future development in this blog post.

If you need to use the old backend in Kotlin 1.5.0, you can add the following lines to the project's configuration file:

- In Gradle:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
 kotlinOptions.useOldBackend = true
}
```

- In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

New default JVM target: 1.8

The default target version for Kotlin/JVM compilations is now 1.8. The 1.6 target is deprecated.

If you need a build for JVM 1.6, you can still switch to this target. Learn how:

- in Gradle
- in Maven
- in the command-line compiler

SAM adapters via invokedynamic

Kotlin 1.5.0 now uses dynamic invocations (invokedynamic) for compiling SAM (Single Abstract Method) conversions:

- Over any expression if the SAM type is a Java interface
- Over lambda if the SAM type is a Kotlin functional interface

The new implementation uses LambdaMetafactory.metafactory() and auxiliary wrapper classes are no longer generated during compilation. This decreases the size of the application's JAR, which improves the JVM startup performance.

To roll back to the old implementation scheme based on anonymous class generation, add the compiler option -Xsam-conversions=class.

Learn how to add compiler options in Gradle, Maven, and the command-line compiler.

Lambdas via invokedynamic

> Compiling plain Kotlin lambdas into invokedynamic is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.

Kotlin 1.5.0 is introducing experimental support for compiling plain Kotlin lambdas (which are not converted to an instance of a functional interface) into dynamic invocations (invokedynamic). The implementation produces lighter binaries by using LambdaMetafactory.metafactory(), which effectively generates the necessary classes at runtime. Currently, it has three limitations compared to ordinary lambda compilation:

- A lambda compiled into invokedynamic is not serializable.
- Calling toString() on such a lambda produces a less readable string representation.
- Experimental reflect API does not support lambdas created with LambdaMetafactory.

To try this feature, add the -Xlambdas=indy compiler option. We would be grateful if you could share your feedback on it using this YouTrack ticket.

Learn how to add compiler options in Gradle, Maven, and command-line compiler.

Deprecation of @JvmDefault and old Xjvm-default modes

Prior to Kotlin 1.4.0, there was the @JvmDefault annotation along with -Xjvm-default=enable and -Xjvm-default=compatibility modes. They served to create the JVM default method for any particular non-abstract member in the Kotlin interface.

In Kotlin 1.4.0, we introduced the new Xjvm-default modes, which switch on default method generation for the whole project.

In Kotlin 1.5.0, we are deprecating @JvmDefault and the old Xjvm-default modes: -Xjvm-default=enable and -Xjvm-default=compatibility.

Learn more about default methods in the Java interop.

Improvements to handling nullability annotations

Kotlin supports handling type nullability information from Java with nullability annotations. Kotlin 1.5.0 introduces a number of improvements for the feature:

- It reads nullability annotations on type arguments in compiled Java libraries that are used as dependencies.
- It supports nullability annotations with the TYPE_USE target for:
  - Arrays
  - Varargs
  - Fields
  - Type parameters and their bounds
  - Type arguments of base classes and interfaces
- If a nullability annotation has multiple targets applicable to a type, and one of these targets is TYPE_USE, then TYPE_USE is preferred. For example, the method signature @Nullable String[] f() becomes fun f(): Array<String?>! if @Nullable supports both TYPE_USE and METHODas targets.

For these newly supported cases, using the wrong type nullability when calling Java from Kotlin produces warnings. Use the -Xtype-enhancement-improvements-strict-mode compiler option to enable strict mode for these cases (with error reporting).

Learn more about null-safety and platform types.

Kotlin/Native

Kotlin/Native is now more performant and stable. The notable changes are:

- Performance improvements
- Deactivation of the memory leak checker

Performance improvements

In 1.5.0, Kotlin/Native is receiving a set of performance improvements that speed up both compilation and execution.

Compiler caches are now supported in debug mode for linuxX64 (only on Linux hosts) and iosArm64 targets. With compiler caches enabled, most debug compilations complete much faster, except for the first one. Measurements showed about a 200% speed increase on our test projects.

To use compiler caches for new targets, opt in by adding the following lines to the project's gradle.properties:

- For linuxX64: kotlin.native.cacheKind.linuxX64=static
- For iosArm64: kotlin.native.cacheKind.iosArm64=static

If you encounter any issues after enabling the compiler caches, please report them to our issue tracker YouTrack.

Other improvements speed up the execution of Kotlin/Native code:

- Trivial property accessors are inlined.
- trimIndent() on string literals is evaluated during the compilation.

Deactivation of the memory leak checker

The built-in Kotlin/Native memory leak checker has been disabled by default.

It was initially designed for internal use, and it is able to find leaks only in a limited number of cases, not all of them. Moreover, it later turned out to have issues that can cause application crashes. So we've decided to turn off the memory leak checker.

The memory leak checker can still be useful for certain cases, for example, unit testing. For these cases, you can enable it by adding the following line of code:

```
Platform.isMemoryLeakCheckerActive = true
```

Note that enabling the checker for the application runtime is not recommended.

Kotlin/JS

Kotlin/JS is receiving evolutionary changes in 1.5.0. We're continuing our work on moving the JS IR compiler backend towards stable and shipping other updates:

- Upgrade of webpack to version 5
- Frameworks and libraries for the IR compiler

Upgrade to webpack 5

The Kotlin/JS Gradle plugin now uses webpack 5 for browser targets instead of webpack 4. This is a major webpack upgrade that brings incompatible changes. If you're using a custom webpack configuration, be sure to check the webpack 5 release notes.

Learn more about bundling Kotlin/JS projects with webpack.

Frameworks and libraries for the IR compiler

> The Kotlin/JS IR compiler is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback on it in YouTrack.

Along with working on the IR-based backend for Kotlin/JS compiler, we encourage and help library authors to build their projects in both mode. This means they are able to produce artifacts for both Kotlin/JS compilers, therefore growing the ecosystem for the new compiler.

Many well-known frameworks and libraries are already available for the IR backend: KVision, fritz2, doodle, and others. If you're using them in your project, you can already build it with the IR backend and see the benefits it brings.

If you're writing your own library, compile it in the 'both' mode so that your clients can also use it with the new compiler.

Kotlin Multiplatform

In Kotlin 1.5.0, choosing a testing dependency for each platform has been simplified and it is now done automatically by the Gradle plugin.

A new API for getting a char category is now available in multiplatform projects.

Standard library

The standard library has received a range of changes and improvements, from stabilizing experimental parts to adding new features:

- Stable unsigned integer types
- Stable locale-agnostic API for uppercase/lowercase text
- Stable Char-to-integer conversion API
- Stable Path API
- Floored division and the mod operator
- Duration API changes
- New API for getting a char category now available in multiplatform code
- New collections function firstNotNullOf()
- Strict version of String?.toBoolean()

You can learn more about the standard library changes in this blog post.

Watch video online.

Stable unsigned integer types

The UInt, ULong, UByte, UShort unsigned integer types are now Stable. The same goes for operations on these types, ranges, and progressions of them. Unsigned arrays and operations on them remain in Beta.

Learn more about unsigned integer types.

Stable locale-agnostic API for upper/lowercasing text

This release brings a new locale-agnostic API for uppercase/lowercase text conversion. It provides an alternative to the toLowerCase(), toUpperCase(), capitalize(), and decapitalize() API functions, which are locale-sensitive. The new API helps you avoid errors due to different locale settings.

Kotlin 1.5.0 provides the following fully Stable alternatives:

- For String functions:

Earlier versions	1.5.0 alternative
String.toUpperCase()	String.uppercase()

Earlier versions	1.5.0 alternative
String.toLowerCase()	String.lowercase()
String.capitalize()	String.replaceFirstChar { it.uppercase() }
String.decapitalize()	String.replaceFirstChar { it.lowercase() }

- For Char functions:

Earlier versions	1.5.0 alternative
Char.toUpperCase()	Char.uppercaseChar(): Char Char.uppercase(): String
Char.toLowerCase()	Char.lowercaseChar(): Char Char.lowercase(): String
Char.toTitleCase()	Char.titlecaseChar(): Char Char.titlecase(): String

> For Kotlin/JVM, there are also overloaded uppercase(), lowercase(), and titlecase() functions with an explicit Locale parameter.

The old API functions are marked as deprecated and will be removed in a future release.

See the full list of changes to the text processing functions in KEEP.

Stable char-to-integer conversion API

Starting from Kotlin 1.5.0, new char-to-code and char-to-digit conversion functions are Stable. These functions replace the current API functions, which were often confused with the similar string-to-Int conversion.

The new API removes this naming confusion, making the code behavior more transparent and unambiguous.

This release introduces Char conversions that are divided into the following sets of clearly named functions:

- Functions to get the integer code of Char and to construct Char from the given code:

```
fun Char(code: Int): Char
fun Char(code: UShort): Char
val Char.code: Int
```

- Functions to convert Char to the numeric value of the digit it represents:

```
fun Char.digitToInt(radix: Int): Int
fun Char.digitToIntOrNull(radix: Int): Int?
```

- An extension function for Int to convert the non-negative single digit it represents to the corresponding Char representation:

```
fun Int.digitToChar(radix: Int): Char
```

The old conversion APIs, including Number.toChar() with its implementations (all except Int.toChar()) and Char extensions for conversion to a numeric type, like Char.toInt(), are now deprecated.

Learn more about the char-to-integer conversion API in KEEP.

Stable Path API

The experimental Path API with extensions for java.nio.file.Path is now Stable.

```
// construct path with the div (/) operator
val baseDir = Path("/base")
val subDir = baseDir / "subdirectory"

// list files in a directory
val kotlinFiles: List<Path> = Path("/home/user").listDirectoryEntries("*.kt")
```

Learn more about the Path API.

Floored division and the mod operator

New operations for modular arithmetics have been added to the standard library:

- floorDiv() returns the result of floored division. It is available for integer types.
- mod() returns the remainder of floored division (modulus). It is available for all numeric types.

These operations look quite similar to the existing division of integers and rem() function (or the %operator), but they work differently on negative numbers:

- a.floorDiv(b) differs from a regular / in that floorDiv rounds the result down (towards the lesser integer), whereas / truncates the result to the integer closer to 0.
- a.mod(b) is the difference between a and a.floorDiv(b) * b. It's either zero or has the same sign as b, while a % b can have a different one.

```
fun main() {
//sampleStart
    println("Floored division -5/3: ${(-5).floorDiv(3)}")
    println( "Modulus: ${(-5).mod(3)}")

    println("Truncated division -5/3: ${-5 / 3}")
    println( "Remainder: ${-5 % 3}")
//sampleEnd
}
```

Duration API changes

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.

There is an experimental Duration class for representing duration amounts in different time units. In 1.5.0, the Duration API has received the following changes:

- Internal value representation now uses Long instead of Double to provide better precision.
- There is a new API for conversion to a particular time unit in Long. It comes to replace the old API, which operates with Double values and is now deprecated. For example, Duration.inWholeMinutes returns the value of the duration expressed as Long and replaces Duration.inMinutes.
- There are new companion functions for constructing a Duration from a number. For example, Duration.seconds(Int) creates a Duration object representing an integer number of seconds. Old extension properties like Int.seconds are now deprecated.

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
//sampleStart
    val duration = Duration.milliseconds(120000)
    println("There are ${duration.inWholeSeconds} seconds in ${duration.inWholeMinutes} minutes")
//sampleEnd
```

```
}
```

New API for getting a char category now available in multiplatform code

Kotlin 1.5.0 introduces the new API for getting a character's category according to Unicode in multiplatform projects. Several functions are now available in all the platforms and in the common code.

Functions for checking whether a char is a letter or a digit:

- Char.isDigit()
- Char.isLetter()
- Char.isLetterOrDigit()

```
fun main() {
//sampleStart
    val chars = listOf('a', '1', '+')
    val (letterOrDigitList, notLetterOrDigitList) = chars.partition { it.isLetterOrDigit() }
    println(letterOrDigitList) // [a, 1]
    println(notLetterOrDigitList) // [+]
//sampleEnd
}
```

Functions for checking the case of a char:

- Char.isLowerCase()
- Char.isUpperCase()
- Char.isTitleCase()

```
fun main() {
//sampleStart
    val chars = listOf('ǅ', 'ǈ', 'ǋ', 'ǲ', '1', 'A', 'a', '+')
    val (titleCases, notTitleCases) = chars.partition { it.isTitleCase() }
    println(titleCases) // [ǅ, ǈ, ǋ, ǲ]
    println(notTitleCases) // [1, A, a, +]
//sampleEnd
}
```

Some other functions:

- Char.isDefined()
- Char.isISOControl()

The property Char.category and its return type enum class CharCategory, which indicates a char's general category according to Unicode, are now also available in multiplatform projects.

Learn more about characters.

New collections function firstNotNullOf()

The new firstNotNullOf() and firstNotNullOfOrNull() functions combine mapNotNull() with first() or firstOrNull(). They map the original collection with the custom selector function and return the first non-null value. If there is no such value, firstNotNullOf() throws an exception, and firstNotNullOfOrNull() returns null.

```
fun main() {
//sampleStart
    val data = listOf("Kotlin", "1.5")
    println(data.firstNotNullOf(String::toDoubleOrNull))
    println(data.firstNotNullOfOrNull(String::toIntOrNull))
//sampleEnd
}
```

Strict version of String?.toBoolean()

Two new functions introduce case-sensitive strict versions of the existing String?.toBoolean():

- String.toBooleanStrict() throws an exception for all inputs except the literals true and false.
- String.toBooleanStrictOrNull() returns null for all inputs except the literals true and false.

```
fun main() {
//sampleStart
    println("true".toBooleanStrict())
    println("1".toBooleanStrictOrNull())
    // println("1".toBooleanStrict()) // Exception
//sampleEnd
}
```

kotlin-test library

The kotlin-test library introduces some new features:

- Simplified test dependencies usage in multiplatform projects
- Automatic selection of a testing framework for Kotlin/JVM source sets
- Assertion function updates

Simplified test dependencies usage in multiplatform projects

Now you can use the kotlin-test dependency to add dependencies for testing in the commonTest source set, and the Gradle plugin will infer the corresponding platform dependencies for each test source set:

- kotlin-test-junit for JVM source sets, see automatic choice of a testing framework for Kotlin/JVM source sets
- kotlin-test-js for Kotlin/JS source sets
- kotlin-test-common and kotlin-test-annotations-common for common source sets
- No extra artifact for Kotlin/Native source sets

Additionally, you can use the kotlin-test dependency in any shared or platform-specific source set.

An existing kotlin-test setup with explicit dependencies will continue to work both in Gradle and in Maven.

Learn more about setting dependencies on test libraries.

Automatic selection of a testing framework for Kotlin/JVM source sets

The Gradle plugin now chooses and adds a dependency on a testing framework automatically. All you need to do is add the dependency kotlin-test in the common source set.

Gradle uses JUnit 4 by default. Therefore, the kotlin("test") dependency resolves to the variant for JUnit 4, namely kotlin-test-junit:

Kotlin

```
kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // This brings the dependency
                                               // on JUnit 4 transitively
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings the dependency
```

```
                                            // on JUnit 4 transitively
            }
        }
    }
}
```

You can choose JUnit 5 or TestNG by calling useJUnitPlatform() or useTestNG() in the test task:

```
tasks {
    test {
        // enable TestNG support
        useTestNG()
        // or
        // enable JUnit Platform (a.k.a. JUnit 5) support
        useJUnitPlatform()
    }
}
```

You can disable automatic testing framework selection by adding the line kotlin.test.infer.jvm.variant=false to the project's gradle.properties.

Learn more about setting dependencies on test libraries.

Assertion function updates

This release brings new assertion functions and improves the existing ones.

The kotlin-test library now has the following features:

- Checking the type of a value

  You can use the new assertIs<T> and assertIsNot<T> to check the type of a value:

  ```
  @Test
  fun testFunction() {
      val s: Any = "test"
      assertIs<String>(s)  // throws AssertionError mentioning the actual type of s if the assertion fails
      // can now print s.length because of contract in assertIs
      println("${s.length}")
  }
  ```

  Because of type erasure, this assert function only checks whether the value is of the List type in the following example and doesn't check whether it's a list of the particular String element type: assertIs<List<String>>(value).

- Comparing the container content for arrays, sequences, and arbitrary iterables

  There is a new set of overloaded assertContentEquals() functions for comparing content for different collections that don't implement structural equality:

  ```
  @Test
  fun test() {
      val expectedArray = arrayOf(1, 2, 3)
      val actualArray = Array(3) { it + 1 }
      assertContentEquals(expectedArray, actualArray)
  }
  ```

- New overloads to assertEquals() and assertNotEquals() for Double and Float numbers

  There are new overloads for the assertEquals() function that make it possible to compare two Double or Float numbers with absolute precision. The precision value is specified as the third parameter of the function:

  ```
   @Test
  fun test() {
      val x = sin(PI)

      // precision parameter
      val tolerance = 0.000001

      assertEquals(0.0, x, tolerance)
  }
  ```

* New functions for checking the content of collections and elements

  You can now check whether the collection or element contains something with the assertContains() function. You can use it with Kotlin collections and elements that have the contains() operator, such as IntRange, String, and others:

```
@Test
fun test() {
    val sampleList = listOf<String>("sample", "sample2")
    val sampleString = "sample"
    assertContains(sampleList, sampleString)  // element in collection
    assertContains(sampleString, "amp")       // substring in string
}
```

* assertTrue(), assertFalse(), expect() functions are now inline

  From now on, you can use these as inline functions, so it's possible to call suspend functions inside a lambda expression:

```
@Test
fun test() = runBlocking<Unit> {
    val deferred = async { "Kotlin is nice" }
    assertTrue("Kotlin substring should be present") {
        deferred.await() .contains("Kotlin")
    }
}
```

kotlinx libraries

Along with Kotlin 1.5.0, we are releasing new versions of the kotlinx libraries:

* kotlinx.coroutines 1.5.0-RC
* kotlinx.serialization 1.2.1
* kotlinx-datetime 0.2.0

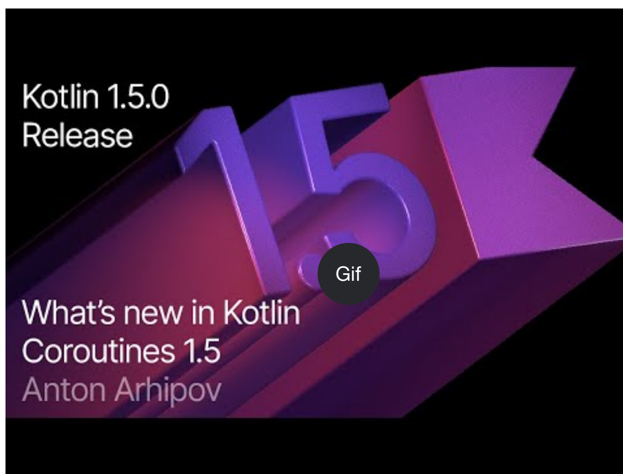
Coroutines 1.5.0-RC

kotlinx.coroutines 1.5.0-RC is here with:

* New channels API
* Stable reactive integrations
* And more

Starting with Kotlin 1.5.0, experimental coroutines are disabled and the -Xcoroutines=experimental flag is no longer supported.

Learn more in the changelog and the kotlinx.coroutines 1.5.0 release blog post.



Watch video online.

Serialization 1.2.1

kotlinx.serialization 1.2.1 is here with:

- Improvements to JSON serialization performance
- Support for multiple names in JSON serialization
- Experimental .proto schema generation from @Serializable classes
- And more

Learn more in the changelog and the kotlinx.serialization 1.2.1 release blog post.

Watch video online.

dateTime 0.2.0

kotlinx-datetime 0.2.0 is here with:

- @Serializable Datetime objects
- Normalized API of DateTimePeriod and DatePeriod
- And more

Learn more in the changelog and the kotlinx-datetime 0.2.0 release blog post.

Migrating to Kotlin 1.5.0

IntelliJ IDEA and Android Studio will suggest updating the Kotlin plugin to 1.5.0 once it is available.

To migrate existing projects to Kotlin 1.5.0, just change the Kotlin version to 1.5.0 and re-import your Gradle or Maven project. Learn how to update to Kotlin 1.5.0.

To start a new project with Kotlin 1.5.0, update the Kotlin plugin and run the Project Wizard from File | New | Project.

The new command-line compiler is available for downloading on the GitHub release page.

Kotlin 1.5.0 is a feature release and therefore can bring incompatible changes to the language. Find the detailed list of such changes in the Compatibility Guide for Kotlin 1.5.

What's new in Kotlin 1.4.30

Released: 3 February 2021

Kotlin 1.4.30 offers preview versions of new language features, promotes the new IR backend of the Kotlin/JVM compiler to Beta, and ships various performance and functional improvements.

You can also learn about new features in this blog post.

Language features

Kotlin 1.5.0 is going to deliver new language features – JVM records support, sealed interfaces, and Stable inline classes. In Kotlin 1.4.30, you can try these features and improvements in preview mode. We would be very grateful if you share your feedback with us in the corresponding YouTrack tickets, as that will allow us to address it before the release of 1.5.0.

- JVM records support
- Sealed interfaces and sealed class improvements
- Improved inline classes

To enable these language features and improvements in preview mode, you need to opt in by adding specific compiler options. See the sections below for details.

Learn more about the new features preview in this blog post.

JVM records support

> The JVM records feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The JDK 16 release includes plans to stabilize a new Java class type called record. To provide all the benefits of Kotlin and maintain its interoperability with Java, Kotlin is introducing experimental record class support.

You can use record classes that are declared in Java just like classes with properties in Kotlin. No additional steps are required.

Starting with 1.4.30, you can declare the record class in Kotlin using the@JvmRecord annotation for a data class:

```
@JvmRecord
data class User(val name: String, val age: Int)
```

To try the preview version of JVM records, add the compiler options -Xjvm-enable-preview and -language-version 1.5.

We're continuing to work on JVM records support, and we would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about implementation, restrictions, and the syntax in KEEP.

Sealed interfaces

> Sealed interfaces are Experimental. They may be dropped or changed at any time. Opt-in is required (see the details below), and you should use them only for evaluation purposes. We would appreciate your feedback on them in YouTrack.

In Kotlin 1.4.30, we're shipping the prototype of sealed interfaces. They complement sealed classes and make it possible to build more flexible restricted class hierarchies.

They can serve as "internal" interfaces that cannot be implemented outside the same module. You can rely on that fact, for example, to write exhaustive when expressions.

```
sealed interface Polygon

class Rectangle(): Polygon
class Triangle(): Polygon

// when() is exhaustive: no other polygon implementations can appear
// after the module is compiled
```

```
fun draw(polygon: Polygon) = when (polygon) {
    is Rectangle -> // ...
    is Triangle -> // ...
}
```

Another use-case: with sealed interfaces, you can inherit a class from two or more sealed superclasses.

```
sealed interface Fillable {
   fun fill()
}
sealed interface Polygon {
   val vertices: List<Point>
}

class Rectangle(override val vertices: List<Point>): Fillable, Polygon {
   override fun fill() { /*...*/ }
}
```

To try the preview version of sealed interfaces, add the compiler option -language-version 1.5. Once you switch to this version, you'll be able to use the sealed modifier on interfaces. We would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about sealed interfaces.

Package-wide sealed class hierarchies

> Package-wide hierarchies of sealed classes are Experimental. They may be dropped or changed at any time. Opt-in is required (see the details below), and you should use them only for evaluation purposes. We would appreciate your feedback on them in YouTrack.

Sealed classes can now form more flexible hierarchies. They can have subclasses in all files of the same compilation unit and the same package. Previously, all subclasses had to appear in the same file.

Direct subclasses may be top-level or nested inside any number of other named classes, named interfaces, or named objects. The subclasses of a sealed class must have a name that is properly qualified – they cannot be local nor anonymous objects.

To try package-wide hierarchies of sealed classes, add the compiler option -language-version 1.5. We would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about package-wide hierarchies of sealed classes.

Improved inline classes

> Inline value classes are in Beta. They are almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make. We would appreciate your feedback on the inline classes feature in YouTrack.

Kotlin 1.4.30 promotes inline classes to Beta and brings the following features and improvements to them:

- Since inline classes are value-based, you can define them using the value modifier. The inline and value modifiers are now equivalent to each other. In future Kotlin versions, we're planning to deprecate the inline modifier.

  From now on, Kotlin requires the @JvmInline annotation before a class declaration for the JVM backend:

  ```
  inline class Name(private val s: String)

  value class Name(private val s: String)

  // For JVM backends
  @JvmInline
  value class Name(private val s: String)
  ```

- Inline classes can have init blocks. You can add code to be executed right after the class is instantiated:

```kotlin
@JvmInline
value class Negative(val x: Int) {
  init {
      require(x < 0) { }
  }
}
```

- Calling functions with inline classes from Java code: before Kotlin 1.4.30, you couldn't call functions that accept inline classes from Java because of mangling. From now on, you can disable mangling manually. To call such functions from Java code, you should add the @JvmName annotation before the function declaration:

```kotlin
inline class UInt(val x: Int)

fun compute(x: Int) { }

@JvmName("computeUInt")
fun compute(x: UInt) { }
```

- In this release, we've changed the mangling scheme for functions to fix the incorrect behavior. These changes led to ABI changes.

  Starting with 1.4.30, the Kotlin compiler uses a new mangling scheme by default. Use the -Xuse-14-inline-classes-mangling-scheme compiler flag to force the compiler to use the old 1.4.0 mangling scheme and preserve binary compatibility.

Kotlin 1.4.30 promotes inline classes to Beta and we are planning to make them Stable in future releases. We'd be very grateful if you would share your feedback with us using this YouTrack ticket.

To try the preview version of inline classes, add the compiler option -Xinline-classes or -language-version 1.5.

Learn more about the mangling algorithm in KEEP.

Learn more about inline classes.

Kotlin/JVM

JVM IR compiler backend reaches Beta

The IR-based compiler backend for Kotlin/JVM, which was presented in 1.4.0 in Alpha, has reached Beta. This is the last pre-stable level before the IR backend becomes the default for the Kotlin/JVM compiler.

We're now dropping the restriction on consuming binaries produced by the IR compiler. Previously, you could use code compiled by the new JVM IR backend only if you had enabled the new backend. Starting from 1.4.30, there is no such limitation, so you can use the new backend to build components for third-party use, such as libraries. Try the Beta version of the new backend and share your feedback in our issue tracker.

To enable the new JVM IR backend, add the following lines to the project's configuration file:

- In Gradle:

  Kotlin

```kotlin
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile::class) {
  kotlinOptions.useIR = true
}
```

  Groovy

```groovy
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useIR = true
}
```

- In Maven:

```xml
<configuration>
    <args>
```

```
        <arg>-Xuse-ir</arg>
    </args>
</configuration>
```

Learn more about the changes that the JVM IR backend brings in this blog post.

Kotlin/Native

Performance improvements

Kotlin/Native has received a variety of performance improvements in 1.4.30, which has resulted in faster compilation times. For example, the time required to rebuild the framework in the Networking and data storage with Kotlin Multiplatform Mobile sample has decreased from 9.5 seconds (in 1.4.10) to 4.5 seconds (in 1.4.30).

Apple watchOS 64-bit simulator target

The x86 simulator target has been deprecated for watchOS since version 7.0. To keep up with the latest watchOS versions, Kotlin/Native has the new target watchosX64 for running the simulator on 64-bit architecture.

Support for Xcode 12.2 libraries

We have added support for the new libraries delivered with Xcode 12.2. You can now use them from Kotlin code.

Kotlin/JS

Lazy initialization of top-level properties

> Lazy initialization of top-level properties is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The IR backend for Kotlin/JS is receiving a prototype implementation of lazy initialization for top-level properties. This reduces the need to initialize all top-level properties when the application starts, and it should significantly improve application start-up times.

We'll keep working on the lazy initialization, and we ask you to try the current prototype and share your thoughts and results in this YouTrack ticket or the #javascript channel in the official Kotlin Slack (get an invite here).

To use the lazy initialization, add the -Xir-property-lazy-initialization compiler option when compiling the code with the JS IR compiler.

Gradle project improvements

Support the Gradle configuration cache

Starting with 1.4.30, the Kotlin Gradle plugin supports the configuration cache feature. It speeds up the build process: once you run the command, Gradle executes the configuration phase and calculates the task graph. Gradle caches the result and reuses it for subsequent builds.

To start using this feature, you can use the Gradle command or set up the IntelliJ based IDE.

Standard library

Locale-agnostic API for upper/lowercasing text

The locale-agnostic API feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

This release introduces the experimental locale-agnostic API for changing the case of strings and characters. The current toLowerCase(), toUpperCase(), capitalize(), decapitalize() API functions are locale-sensitive. This means that different platform locale settings can affect code behavior. For example, in the Turkish locale, when the string "kotlin" is converted using toUpperCase, the result is "KOTLİN", not "KOTLIN".

```
// current API
println("Needs to be capitalized".toUpperCase()) // NEEDS TO BE CAPITALIZED

// new API
println("Needs to be capitalized".uppercase()) // NEEDS TO BE CAPITALIZED
```

Kotlin 1.4.30 provides the following alternatives:

- For String functions:

Earlier versions	1.4.30 alternative
String.toUpperCase()	String.uppercase()
String.toLowerCase()	String.lowercase()
String.capitalize()	String.replaceFirstChar { it.uppercase() }
String.decapitalize()	String.replaceFirstChar { it.lowercase() }

- For Char functions:

Earlier versions	1.4.30 alternative
Char.toUpperCase()	Char.uppercaseChar(): Char Char.uppercase(): String
Char.toLowerCase()	Char.lowercaseChar(): Char Char.lowercase(): String
Char.toTitleCase()	Char.titlecaseChar(): Char Char.titlecase(): String

For Kotlin/JVM, there are also overloaded uppercase(), lowercase(), and titlecase() functions with an explicit Locale parameter.

See the full list of changes to the text processing functions in KEEP.

Clear Char-to-code and Char-to-digit conversions

> The unambiguous API for the Char conversion feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The current Char to numbers conversion functions, which return UTF-16 codes expressed in different numeric types, are often confused with the similar String-to-Int conversion, which returns the numeric value of a string:

```
"4".toInt() // returns 4
'4'.toInt() // returns 52
// and there was no common function that would return the numeric value 4 for Char '4'
```

To avoid this confusion we've decided to separate Char conversions into two following sets of clearly named functions:

- Functions to get the integer code of Char and to construct Char from the given code:

```
fun Char(code: Int): Char
fun Char(code: UShort): Char
val Char.code: Int
```

- Functions to convert Char to the numeric value of the digit it represents:

```
fun Char.digitToInt(radix: Int): Int
fun Char.digitToIntOrNull(radix: Int): Int?
```

- An extension function for Int to convert the non-negative single digit it represents to the corresponding Char representation:

```
fun Int.digitToChar(radix: Int): Char
```

See more details in KEEP.

Serialization updates

Along with Kotlin 1.4.30, we are releasing kotlinx.serialization 1.1.0-RC, which includes some new features:

- Inline classes serialization support
- Unsigned primitive type serialization support

Inline classes serialization support

Starting with Kotlin 1.4.30, you can make inline classes serializable:

```
@Serializable
inline class Color(val rgb: Int)
```

> The feature requires the new 1.4.30 IR compiler.

The serialization framework does not box serializable inline classes when they are used in other serializable classes.

Learn more in the kotlinx.serialization docs.

Unsigned primitive type serialization support

Starting from 1.4.30, you can use standard JSON serializers of kotlinx.serialization for unsigned primitive types: UInt, ULong, UByte, and UShort:

```
@Serializable
class Counter(val counted: UByte, val description: String)
fun main() {
   val counted = 239.toUByte()
   println(Json.encodeToString(Counter(counted, "tries")))
```

```
}
```

Learn more in the kotlinx.serialization docs.

What's new in Kotlin 1.4.20

Released: 23 November 2020

Kotlin 1.4.20 offers a number of new experimental features and provides fixes and improvements for existing features, including those added in 1.4.0.

You can also learn about new features with more examples in this blog post.

Kotlin/JVM

Improvements of Kotlin/JVM are intended to keep it up with the features of modern Java versions:

- Java 15 target
- invokedynamic string concatenation

Java 15 target

Now Java 15 is available as a Kotlin/JVM target.

invokedynamic string concatenation

> invokedynamic string concatenation is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin 1.4.20 can compile string concatenations into dynamic invocations on JVM 9+ targets, therefore improving the performance.

Currently, this feature is experimental and covers the following cases:

- String.plus in the operator (a + b), explicit (a.plus(b)), and reference ((a::plus)(b)) form.
- toString on inline and data classes.
- string templates except for ones with a single non-constant argument (see KT-42457).

To enable invokedynamic string concatenation, add the -Xstring-concat compiler option with one of the following values:

- indy-with-constants to perform invokedynamic concatenation on strings with StringConcatFactory.makeConcatWithConstants().
- indy to perform invokedynamic concatenation on strings with StringConcatFactory.makeConcat().
- inline to switch back to the classic concatenation via StringBuilder.append().

Kotlin/JS

Kotlin/JS keeps evolving fast, and in 1.4.20 you can find a number experimental features and improvements:

- Gradle DSL changes
- New Wizard templates
- Ignoring compilation errors with IR compiler

Gradle DSL changes

The Gradle DSL for Kotlin/JS receives a number of updates which simplify project setup and customization. This includes webpack configuration adjustments,

modifications to the auto-generated package.json file, and improved control over transitive dependencies.

Single point for webpack configuration

A new configuration block commonWebpackConfig is available for the browser target. Inside it, you can adjust common settings from a single point, instead of having to duplicate configurations for the webpackTask, runTask, and testTask.

To enable CSS support by default for all three tasks, add the following snippet in the build.gradle(.kts) of your project:

```
browser {
    commonWebpackConfig {
        cssSupport.enabled = true
    }
    binaries.executable()
}
```

Learn more about configuring webpack bundling.

package.json customization from Gradle

For more control over your Kotlin/JS package management and distribution, you can now add properties to the project file package.json via the Gradle DSL.

To add custom fields to your package.json, use the customField function in the compilation's packageJson block:

```
kotlin {
    js(BOTH) {
        compilations["main"].packageJson {
            customField("hello", mapOf("one" to 1, "two" to 2))
        }
    }
}
```

Learn more about package.json customization.

Selective yarn dependency resolutions

> Support for selective yarn dependency resolutions is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin 1.4.20 provides a way of configuring Yarn's selective dependency resolutions - the mechanism for overriding dependencies of the packages you depend on.

You can use it through the YarnRootExtension inside the YarnPlugin in Gradle. To affect the resolved version of a package for your project, use the resolution function passing in the package name selector (as specified by Yarn) and the version to which it should resolve.

```
rootProject.plugins.withType<YarnPlugin> {
    rootProject.the<YarnRootExtension>().apply {
        resolution("react", "16.0.0")
        resolution("processor/decamelize", "3.0.0")
    }
}
```

Here, all of your npm dependencies which require react will receive version 16.0.0, and processor will receive its dependency decamelize as version 3.0.0.

Disabling granular workspaces

> Disabling granular workspaces is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

To speed up build times, the Kotlin/JS Gradle plugin only installs the dependencies which are required for a particular Gradle task. For example, the webpack-dev-server package is only installed when you execute one of the *Run tasks, and not when you execute the assemble task. Such behavior can potentially bring problems when you run multiple Gradle processes in parallel. When the dependency requirements clash, the two installations of npm packages can cause errors.

To resolve this issue, Kotlin 1.4.20 includes an option to disable these so-called granular workspaces. This feature is currently available through the YarnRootExtension inside the YarnPlugin in Gradle. To use it, add the following snippet to your build.gradle.kts file:

```
rootProject.plugins.withType<YarnPlugin> {
    rootProject.the<YarnRootExtension>().disableGranularWorkspaces()
}
```

New Wizard templates

To give you more convenient ways to customize your project during creation, the project wizard for Kotlin comes with new templates for Kotlin/JS applications:

- Browser Application - a minimal Kotlin/JS Gradle project that runs in the browser.
- React Application - a React app that uses the appropriate kotlin-wrappers. It provides options to enable integrations for style-sheets, navigational components, or state containers.
- Node.js Application - a minimal project for running in a Node.js runtime. It comes with the option to directly include the experimental kotlinx-nodejs package.

Ignoring compilation errors with IR compiler

> Ignore compilation errors mode is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

The IR compiler for Kotlin/JS comes with a new experimental mode - compilation with errors. In this mode, you can run you code even if it contains errors, for example, if you want to try certain things it when the whole application is not ready yet.

There are two tolerance policies for this mode:

- SEMANTIC: the compiler will accept code which is syntactically correct, but doesn't make sense semantically, such as val x: String = 3.
- SYNTAX: the compiler will accept any code, even if it contains syntax errors.

To allow compilation with errors, add the -Xerror-tolerance-policy= compiler option with one of the values listed above.

Learn more about ignoring compilation errors with Kotlin/JS IR compiler.

Kotlin/Native

Kotlin/Native's priorities in 1.4.20 are performance and polishing existing features. These are the notable improvements:

- Escape analysis
- Performance improvements and bug fixes
- Opt-in wrapping of Objective-C exceptions
- CocoaPods plugin improvements
- Support for Xcode 12 libraries

Escape analysis

> The escape analysis mechanism is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin/Native receives a prototype of the new escape analysis mechanism. It improves the runtime performance by allocating certain objects on the stack instead of the heap. This mechanism shows a 10% average performance increase on our benchmarks, and we continue improving it so that it speeds up the program even more.

The escape analysis runs in a separate compilation phase for the release builds (with the -opt compiler option).

If you want to disable the escape analysis phase, use the -Xdisable-phases=EscapeAnalysis compiler option.

Performance improvements and bug fixes

Kotlin/Native receives performance improvements and bug fixes in various components, including the ones added in 1.4.0, for example, the code sharing mechanism.

Opt-in wrapping of Objective-C exceptions

> The Objective-C exception wrapping mechanism is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin/Native now can handle exceptions thrown from Objective-C code in runtime to avoid program crashes.

You can opt in to wrap NSException's into Kotlin exceptions of type ForeignException. They hold the references to the original NSException's. This lets you get the information about the root cause and handle it properly.

To enable wrapping of Objective-C exceptions, specify the -Xforeign-exception-mode objc-wrap option in the cinterop call or add foreignExceptionMode = objc-wrap property to .def file. If you use CocoaPods integration, specify the option in the pod {} build script block of a dependency like this:

```
pod("foo") {
    extraOpts = listOf("-Xforeign-exception-mode", "objc-wrap")
}
```

The default behavior remains unchanged: the program terminates when an exception is thrown from the Objective-C code.

CocoaPods plugin improvements

Kotlin 1.4.20 continues the set of improvements in CocoaPods integration. Namely, you can try the following new features:

- Improved task execution
- Extended DSL
- Updated integration with Xcode

Improved task execution

CocoaPods plugin gets an improved task execution flow. For example, if you add a new CocoaPods dependency, existing dependencies are not rebuilt. Adding an extra target also doesn't affect rebuilding dependencies for existing ones.

Extended DSL

The DSL of adding CocoaPods dependencies to your Kotlin project receives new capabilites.

In addition to local Pods and Pods from the CocoaPods repository, you can add dependencies on the following types of libraries:

- A library from a custom spec repository.
- A remote library from a Git repository.
- A library from an archive (also available by arbitrary HTTP address).
- A static library.
- A library with custom cinterop options.

Learn more about adding CocoaPods dependencies in Kotlin projects. Find examples in the Kotlin with CocoaPods sample.

Updated integration with Xcode

To work correctly with Xcode, Kotlin requires some Podfile changes:

- If your Kotlin Pod has any Git, HTTP, or specRepo Pod dependency, you should also specify it in the Podfile.

- When you add a library from the custom spec, you also should specify the location of specs at the beginning of your Podfile.

Now integration errors have a detailed description in IDEA. So if you have problems with your Podfile, you will immediately know how to fix them.

Learn more about creating Kotlin pods.

Support for Xcode 12 libraries

We have added support for new libraries delivered with Xcode 12. Now you can use them from the Kotlin code.

Kotlin Multiplatform

Updated structure of multiplatform library publications

Starting from Kotlin 1.4.20, there is no longer a separate metadata publication. Metadata artifacts are now included in the root publication which stands for the whole library and is automatically resolved to the appropriate platform-specific artifacts when added as a dependency to the common source set.

Learn more about publishing a multiplatform library.

Compatibility with earlier versions

This change of structure breaks the compatibility between projects with hierarchical project structure. If a multiplatform project and a library it depends on both have the hierarchical project structure, then you need to update them to Kotlin 1.4.20 or higher simultaneously. Libraries published with Kotlin 1.4.20 are not available for using from project published with earlier versions.

Projects and libraries without the hierarchical project structure remain compatible.

Standard library

The standard library of Kotlin 1.4.20 offers new extensions for working with files and a better performance.

- Extensions for java.nio.file.Path
- Improved String.replace function performance

Extensions for java.nio.file.Path

> Extensions for java.nio.file.Path are Experimental. They may be dropped or changed at any time. Opt-in is required (see details below). Use them only for evaluation purposes. We appreciate your feedback on them in YouTrack.

Now the standard library provides experimental extensions for java.nio.file.Path. Working with the modern JVM file API in an idiomatic Kotlin way is now similar to working with java.io.File extensions from the kotlin.io package.

```
// construct path with the div (/) operator
val baseDir = Path("/base")
val subDir = baseDir / "subdirectory"

// list files in a directory
val kotlinFiles: List<Path> = Path("/home/user").listDirectoryEntries("*.kt")
```

The extensions are available in the kotlin.io.path package in the kotlin-stdlib-jdk7 module. To use the extensions, opt-in to the experimental annotation @ExperimentalPathApi.

Improved String.replace function performance

The new implementation of String.replace() speeds up the function execution. The case-sensitive variant uses a manual replacement loop based on indexOf, while the case-insensitive one uses regular expression matching.

Kotlin Android Extensions

In 1.4.20 the Kotlin Android Extensions plugin becomes deprecated and Parcelable implementation generator moves to a separate plugin.

- Deprecation of synthetic views
- New plugin for Parcelable implementation generator

Deprecation of synthetic views

Synthetic views were presented in the Kotlin Android Extensions plugin a while ago to simplify the interaction with UI elements and reduce boilerplate. Now Google offers a native mechanism that does the same - Android Jetpack's view bindings, and we're deprecating synthetic views in favor of those.

We extract the Parcelable implementations generator from kotlin-android-extensions and start the deprecation cycle for the rest of it - synthetic views. For now, they will keep working with a deprecation warning. In the future, you'll need to switch your project to another solution. Here are the guidelines that will help you migrate your Android project from synthetics to view bindings.

New plugin for Parcelable implementation generator

The Parcelable implementation generator is now available in the new kotlin-parcelize plugin. Apply this plugin instead of kotlin-android-extensions.

> kotlin-parcelize and kotlin-android-extensions can't be applied together in one module.

The @Parcelize annotation is moved to the kotlinx.parcelize package.

Learn more about Parcelable implementation generator in the Android documentation.

What's new in Kotlin 1.4.0

Released: 17 August 2020

In Kotlin 1.4.0, we ship a number of improvements in all of its components, with the focus on quality and performance. Below you will find the list of the most important changes in Kotlin 1.4.0.

Language features and improvements

Kotlin 1.4.0 comes with a variety of different language features and improvements. They include:

- SAM conversions for Kotlin interfaces
- Explicit API mode for library authors
- Mixing named and positional arguments
- Trailing comma
- Callable reference improvements
- break and continue inside when included in loops

SAM conversions for Kotlin interfaces

Before Kotlin 1.4.0, you could apply SAM (Single Abstract Method) conversions only when working with Java methods and Java interfaces from Kotlin. From now on, you can use SAM conversions for Kotlin interfaces as well. To do so, mark a Kotlin interface explicitly as functional with the fun modifier.

SAM conversion applies if you pass a lambda as an argument when an interface with only one single abstract method is expected as a parameter. In this case, the compiler automatically converts the lambda to an instance of the class that implements the abstract member function.

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}
```

```kotlin
val isEven = IntPredicate { it % 2 == 0 }

fun main() {
    println("Is 7 even? - ${isEven.accept(7)}")
}
```

Learn more about Kotlin functional interfaces and SAM conversions.

Explicit API mode for library authors

Kotlin compiler offers explicit API mode for library authors. In this mode, the compiler performs additional checks that help make the library's API clearer and more consistent. It adds the following requirements for declarations exposed to the library's public API:

- Visibility modifiers are required for declarations if the default visibility exposes them to the public API. This helps ensure that no declarations are exposed to the public API unintentionally.
- Explicit type specifications are required for properties and functions that are exposed to the public API. This guarantees that API users are aware of the types of API members they use.

Depending on your configuration, these explicit APIs can produce errors (strict mode) or warnings (warning mode). Certain kinds of declarations are excluded from such checks for the sake of readability and common sense:

- primary constructors
- properties of data classes
- property getters and setters
- override methods

Explicit API mode analyzes only the production sources of a module.

To compile your module in the explicit API mode, add the following lines to your Gradle build script:

Kotlin

```kotlin
kotlin {
    // for strict mode
    explicitApi()
    // or
    explicitApi = ExplicitApiMode.Strict

    // for warning mode
    explicitApiWarning()
    // or
    explicitApi = ExplicitApiMode.Warning
}
```

Groovy

```groovy
kotlin {
    // for strict mode
    explicitApi()
    // or
    explicitApi = 'strict'

    // for warning mode
    explicitApiWarning()
    // or
    explicitApi = 'warning'
}
```

When using the command-line compiler, switch to explicit API mode by adding the -Xexplicit-api compiler option with the value strict or warning.

```
-Xexplicit-api={strict|warning}
```

Find more details about the explicit API mode in the KEEP.

Mixing named and positional arguments

In Kotlin 1.3, when you called a function with named arguments, you had to place all the arguments without names (positional arguments) before the first named argument. For example, you could call f(1, y = 2), but you couldn't call f(x = 1, 2).

It was really annoying when all the arguments were in their correct positions but you wanted to specify a name for one argument in the middle. It was especially helpful for making absolutely clear which attribute a boolean or null value belongs to.

In Kotlin 1.4, there is no such limitation – you can now specify a name for an argument in the middle of a set of positional arguments. Moreover, you can mix positional and named arguments any way you like, as long as they remain in the correct order.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Char = ' '
) {
    // ...
}

//Function call with a named argument in the middle
reformat("This is a String!", uppercaseFirstLetter = false , '-')
```

Trailing comma

With Kotlin 1.4 you can now add a trailing comma in enumerations such as argument and parameter lists, when entries, and components of destructuring declarations. With a trailing comma, you can add new items and change their order without adding or removing commas.

This is especially helpful if you use multi-line syntax for parameters or values. After adding a trailing comma, you can then easily swap lines with parameters or values.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Character = ' ', //trailing comma
) {
    // ...
}
```

```
val colors = listOf(
    "red",
    "green",
    "blue", //trailing comma
)
```

Callable reference improvements

Kotlin 1.4 supports more cases for using callable references:

- References to functions with default argument values
- Function references in Unit-returning functions
- References that adapt based on the number of arguments in a function
- Suspend conversion on callable references

References to functions with default argument values

Now you can use callable references to functions with default argument values. If the callable reference to the function foo takes no arguments, the default value 0 is used.

```
fun foo(i: Int = 0): String = "$i!"

fun apply(func: () -> String): String = func()

fun main() {
    println(apply(::foo))
}
```

Previously, you had to write additional overloads for the function apply to use the default argument values.

```
// some new overload
fun applyInt(func: (Int) -> String): String = func(0)
```

Function references in Unit-returning functions

In Kotlin 1.4, you can use callable references to functions returning any type in Unit-returning functions. Before Kotlin 1.4, you could only use lambda arguments in this case. Now you can use both lambda arguments and callable references.

```
fun foo(f: () -> Unit) { }
fun returnsInt(): Int = 42

fun main() {
    foo { returnsInt() } // this was the only way to do it  before 1.4
    foo(::returnsInt) // starting from 1.4, this also works
}
```

References that adapt based on the number of arguments in a function

Now you can adapt callable references to functions when passing a variable number of arguments (vararg) . You can pass any number of parameters of the same type at the end of the list of passed arguments.

```
fun foo(x: Int, vararg y: String) {}

fun use0(f: (Int) -> Unit) {}
fun use1(f: (Int, String) -> Unit) {}
fun use2(f: (Int, String, String) -> Unit) {}

fun test() {
    use0(::foo)
    use1(::foo)
    use2(::foo)
}
```

Suspend conversion on callable references

In addition to suspend conversion on lambdas, Kotlin now supports suspend conversion on callable references starting from version 1.4.0.

```
fun call() {}
fun takeSuspend(f: suspend () -> Unit) {}

fun test() {
    takeSuspend { call() } // OK before 1.4
    takeSuspend(::call) // In Kotlin 1.4, it also works
}
```

Using break and continue inside when expressions included in loops

In Kotlin 1.3, you could not use unqualified break and continue inside when expressions included in loops. The reason was that these keywords were reserved for possible fall-through behavior in when expressions.

That's why if you wanted to use break and continue inside when expressions in loops, you had to label them, which became rather cumbersome.

```
fun test(xs: List<Int>) {
    LOOP@for (x in xs) {
        when (x) {
            2 -> continue@LOOP
            17 -> break@LOOP
            else -> println(x)
        }
    }
}
```

In Kotlin 1.4, you can use break and continue without labels inside when expressions included in loops. They behave as expected by terminating the nearest enclosing loop or proceeding to its next step.

```
fun test(xs: List<Int>) {
    for (x in xs) {
        when (x) {
            2 -> continue
            17 -> break
            else -> println(x)
        }
    }
}
```

The fall-through behavior inside when is subject to further design.

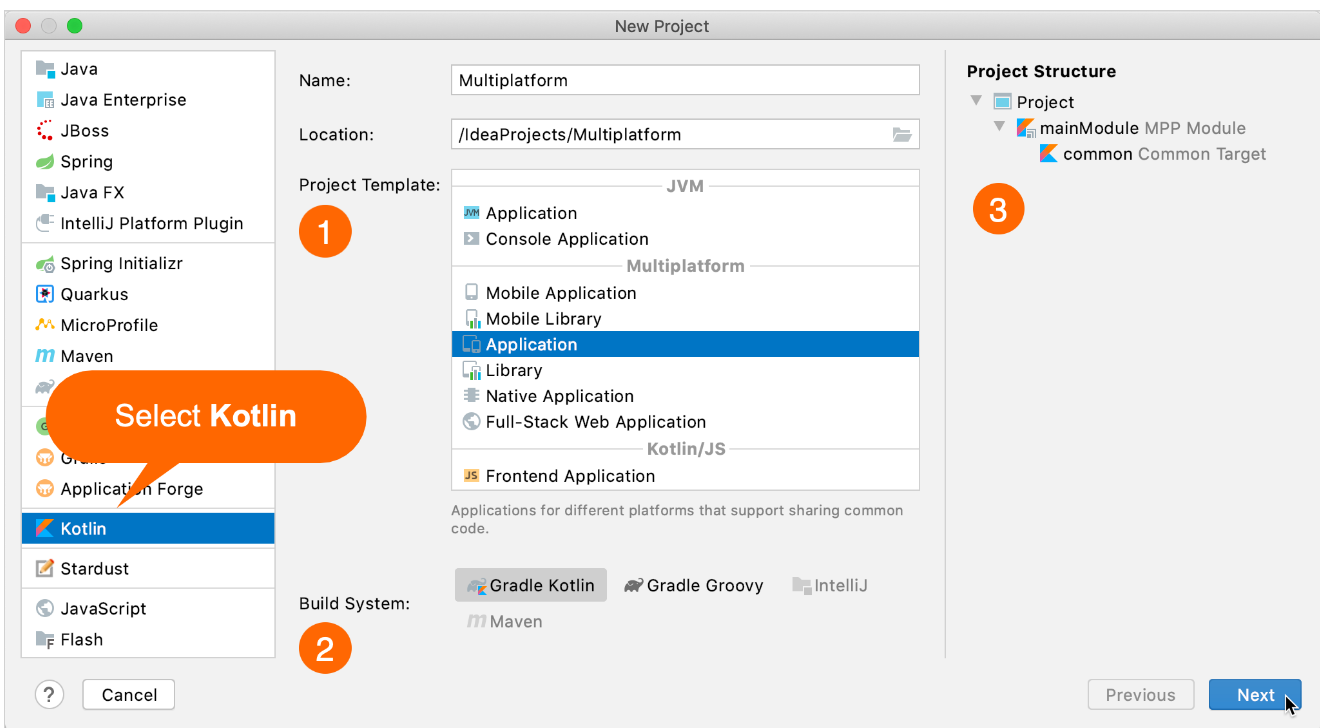
New tools in the IDE

With Kotlin 1.4, you can use the new tools in IntelliJ IDEA to simplify Kotlin development:

- New flexible Project Wizard
- Coroutine Debugger

New flexible Project Wizard

With the flexible new Kotlin Project Wizard, you have a place to easily create and configure different types of Kotlin projects, including multiplatform projects, which can be difficult to configure without a UI.



Kotlin Project Wizard – Multiplatform project

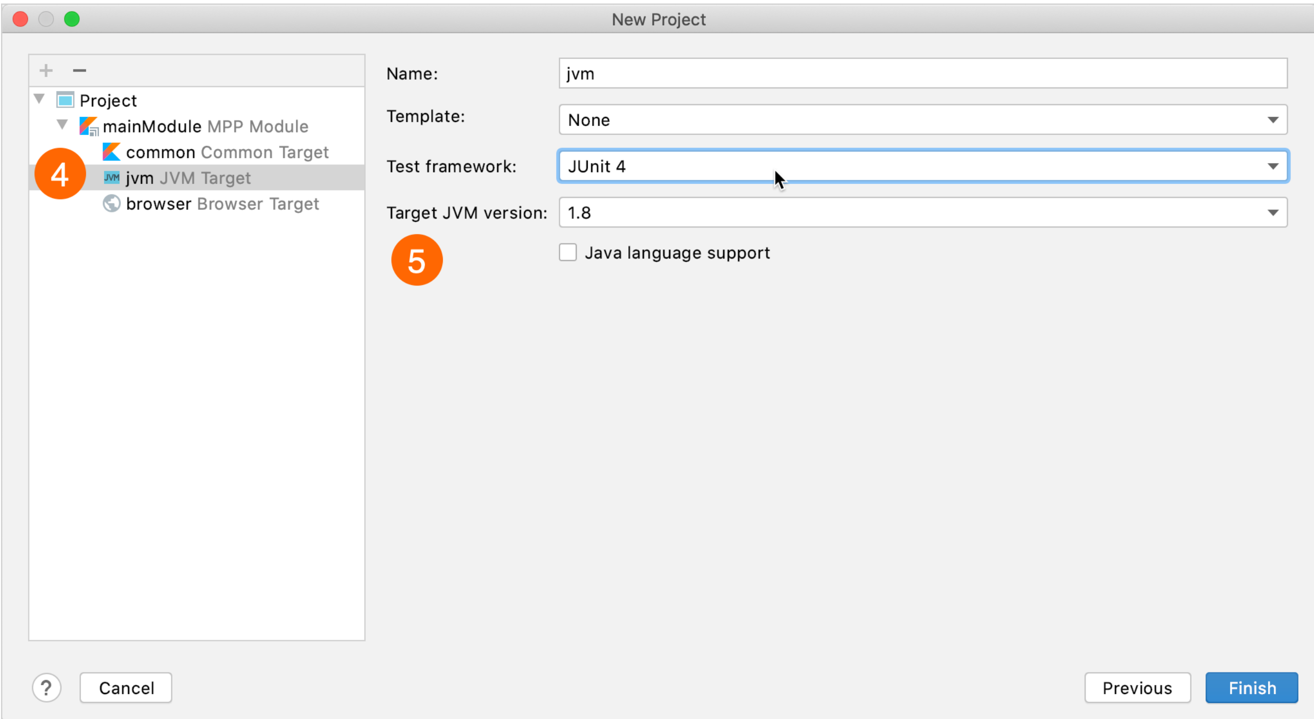
The new Kotlin Project Wizard is both simple and flexible:

1. Select the project template, depending on what you're trying to do. More templates will be added in the future.
2. Select the build system – Gradle (Kotlin or Groovy DSL), Maven, or IntelliJ IDEA.
   The Kotlin Project Wizard will only show the build systems supported on the selected project template.
3. Preview the project structure directly on the main screen.

Then you can finish creating your project or, optionally, configure the project on the next screen:

4. Add/remove modules and targets supported for this project template.

5. Configure module and target settings, for example, the target JVM version, target template, and test framework.



Kotlin Project Wizard - Configure targets

In the future, we are going to make the Kotlin Project Wizard even more flexible by adding more configuration options and templates.

You can try out the new Kotlin Project Wizard by working through these tutorials:

- Create a console application based on Kotlin/JVM
- Create a Kotlin/JS application for React
- Create a Kotlin/Native application

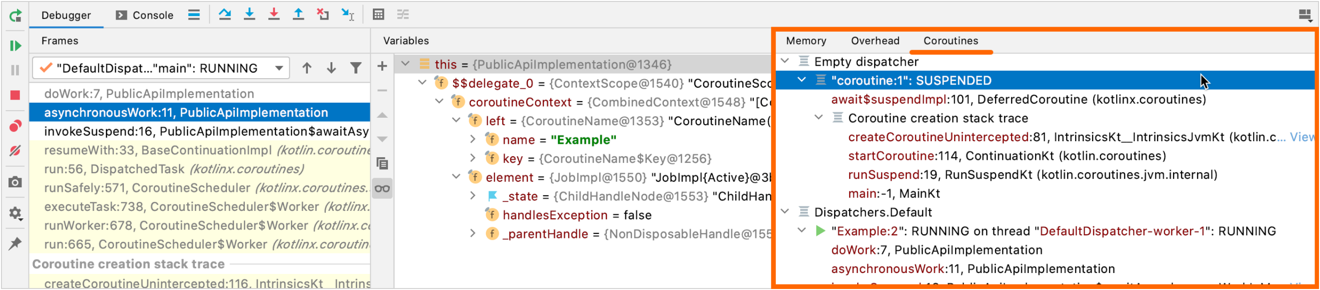
Coroutine Debugger

Many people already use coroutines for asynchronous programming. But when it came to debugging, working with coroutines before Kotlin 1.4, could be a real pain. Since coroutines jumped between threads, it was difficult to understand what a specific coroutine was doing and check its context. In some cases, tracking steps over breakpoints simply didn't work. As a result, you had to rely on logging or mental effort to debug code that used coroutines.

In Kotlin 1.4, debugging coroutines is now much more convenient with the new functionality shipped with the Kotlin plugin.

> Debugging works for versions 1.3.8 or later of kotlinx-coroutines-core.

The Debug Tool Window now contains a new Coroutines tab. In this tab, you can find information about both currently running and suspended coroutines. The coroutines are grouped by the dispatcher they are running on.

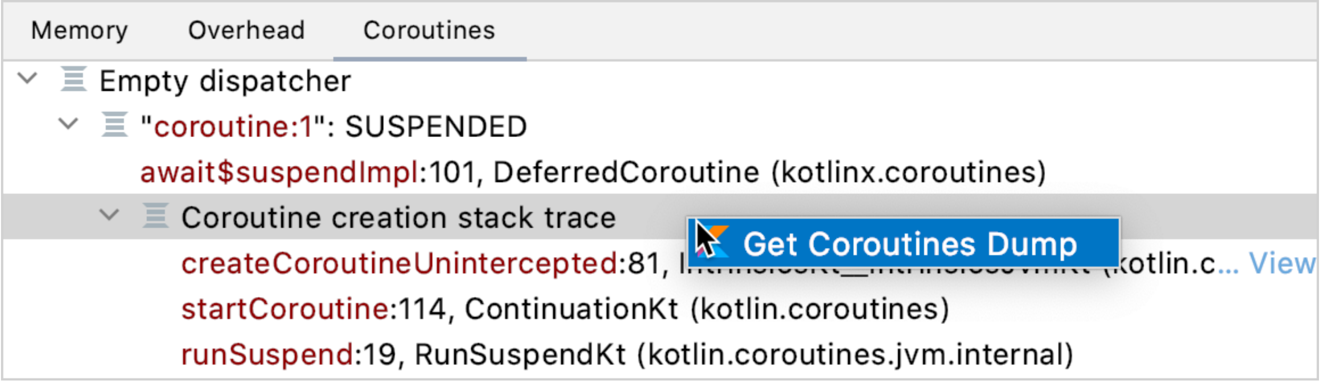


Debugging coroutines

Now you can:

- Easily check the state of each coroutine.
- See the values of local and captured variables for both running and suspended coroutines.
- See a full coroutine creation stack, as well as a call stack inside the coroutine. The stack includes all frames with variable values, even those that would be lost during standard debugging.

If you need a full report containing the state of each coroutine and its stack, right-click inside the Coroutines tab, and then click Get Coroutines Dump. Currently, the coroutines dump is rather simple, but we're going to make it more readable and helpful in future versions of Kotlin.



Coroutines Dump

Learn more about debugging coroutines in this blog post and IntelliJ IDEA documentation.

New compiler

The new Kotlin compiler is going to be really fast; it will unify all the supported platforms and provide an API for compiler extensions. It's a long-term project, and we've already completed several steps in Kotlin 1.4.0:

- New, more powerful type inference algorithm is enabled by default.
- New JVM and JS IR backends. They will become the default once we stabilize them.

New more powerful type inference algorithm

Kotlin 1.4 uses a new, more powerful type inference algorithm. This new algorithm was already available to try in Kotlin 1.3 by specifying a compiler option, and now it's used by default. You can find the full list of issues fixed in the new algorithm in YouTrack. Here you can find some of the most noticeable improvements:

- More cases where type is inferred automatically
- Smart casts for a lambda's last expression
- Smart casts for callable references
- Better inference for delegated properties

- SAM conversion for Java interfaces with different arguments
- Java SAM interfaces in Kotlin

More cases where type is inferred automatically

The new inference algorithm infers types for many cases where the old algorithm required you to specify them explicitly. For instance, in the following example the type of the lambda parameter it is correctly inferred to String?:

```
//sampleStart
val rulesMap: Map<String, (String?) -> Boolean> = mapOf(
    "weak" to { it != null },
    "medium" to { !it.isNullOrBlank() },
    "strong" to { it != null && "^[a-zA-Z0-9]+$".toRegex().matches(it) }
)
//sampleEnd

fun main() {
    println(rulesMap.getValue("weak")("abc!"))
    println(rulesMap.getValue("strong")("abc"))
    println(rulesMap.getValue("strong")("abc!"))
}
```

In Kotlin 1.3, you needed to introduce an explicit lambda parameter or replace to with a Pair constructor with explicit generic arguments to make it work.

Smart casts for a lambda's last expression

In Kotlin 1.3, the last expression inside a lambda wasn't smart cast unless you specified the expected type. Thus, in the following example, Kotlin 1.3 infers String? as the type of the result variable:

```
val result = run {
    var str = currentValue()
    if (str == null) {
        str = "test"
    }
    str // the Kotlin compiler knows that str is not null here
}
// The type of 'result' is String? in Kotlin 1.3 and String in Kotlin 1.4
```

In Kotlin 1.4, thanks to the new inference algorithm, the last expression inside a lambda gets smart cast, and this new, more precise type is used to infer the resulting lambda type. Thus, the type of the result variable becomes String.

In Kotlin 1.3, you often needed to add explicit casts (either !! or type casts like as String) to make such cases work, and now these casts have become unnecessary.

Smart casts for callable references

In Kotlin 1.3, you couldn't access a member reference of a smart cast type. Now in Kotlin 1.4 you can:

```
import kotlin.reflect.KFunction

sealed class Animal
class Cat : Animal() {
    fun meow() {
        println("meow")
    }
}

class Dog : Animal() {
    fun woof() {
        println("woof")
    }
}

//sampleStart
fun perform(animal: Animal) {
    val kFunction: KFunction<*> = when (animal) {
        is Cat -> animal::meow
        is Dog -> animal::woof
    }
    kFunction.call()
}
//sampleEnd
```

```
fun main() {
    perform(Cat())
}
```

You can use different member references animal::meow and animal::woof after the animal variable has been smart cast to specific types Cat and Dog. After type checks, you can access member references corresponding to subtypes.

Better inference for delegated properties

The type of a delegated property wasn't taken into account while analyzing the delegate expression which follows the by keyword. For instance, the following code didn't compile before, but now the compiler correctly infers the types of the old and new parameters as String?:

```
import kotlin.properties.Delegates

fun main() {
    var prop: String? by Delegates.observable(null) { p, old, new ->
        println("$old → $new")
    }
    prop = "abc"
    prop = "xyz"
}
```

SAM conversion for Java interfaces with different arguments

Kotlin has supported SAM conversions for Java interfaces from the beginning, but there was one case that wasn't supported, which was sometimes annoying when working with existing Java libraries. If you called a Java method that took two SAM interfaces as parameters, both arguments needed to be either lambdas or regular objects. You couldn't pass one argument as a lambda and another as an object.

The new algorithm fixes this issue, and you can pass a lambda instead of a SAM interface in any case, which is the way you'd naturally expect it to work.

```
// FILE: A.java
public class A {
    public static void foo(Runnable r1, Runnable r2) {}
}
```

```
// FILE: test.kt
fun test(r1: Runnable) {
    A.foo(r1) {}  // Works in Kotlin 1.4
}
```

Java SAM interfaces in Kotlin

In Kotlin 1.4, you can use Java SAM interfaces in Kotlin and apply SAM conversions to them.

```
import java.lang.Runnable

fun foo(r: Runnable) {}

fun test() {
    foo { } // OK
}
```

In Kotlin 1.3, you would have had to declare the function foo above in Java code to perform a SAM conversion.

Unified backends and extensibility

In Kotlin, we have three backends that generate executables: Kotlin/JVM, Kotlin/JS, and Kotlin/Native. Kotlin/JVM and Kotlin/JS don't share much code since they were developed independently of each other. Kotlin/Native is based on a new infrastructure built around an intermediate representation (IR) for Kotlin code.

We are now migrating Kotlin/JVM and Kotlin/JS to the same IR. As a result, all three backends share a lot of logic and have a unified pipeline. This allows us to implement most features, optimizations, and bug fixes only once for all platforms. Both new IR-based back-ends are in Alpha.

A common backend infrastructure also opens the door for multiplatform compiler extensions. You will be able to plug into the pipeline and add custom processing and transformations that will automatically work for all platforms.

We encourage you to use our new JVM IR and JS IR backends, which are currently in Alpha, and share your feedback with us.

Kotlin/JVM

Kotlin 1.4.0 includes a number of JVM-specific improvements, such as:

- New JVM IR backend
- New modes for generating default methods in interfaces
- Unified exception type for null checks
- Type annotations in the JVM bytecode

New JVM IR backend

Along with Kotlin/JS, we are migrating Kotlin/JVM to the unified IR backend, which allows us to implement most features and bug fixes once for all platforms. You will also be able to benefit from this by creating multiplatform extensions that will work for all platforms.

Kotlin 1.4.0 does not provide a public API for such extensions yet, but we are working closely with our partners, including Jetpack Compose, who are already building their compiler plugins using our new backend.

We encourage you to try out the new Kotlin/JVM backend, which is currently in Alpha, and to file any issues and feature requests to our issue tracker. This will help us to unify the compiler pipelines and bring compiler extensions like Jetpack Compose to the Kotlin community more quickly.

To enable the new JVM IR backend, specify an additional compiler option in your Gradle build script:

```
kotlinOptions.useIR = true
```

> If you enable Jetpack Compose, you will automatically be opted in to the new JVM backend without needing to specify the compiler option in kotlinOptions.

When using the command-line compiler, add the compiler option -Xuse-ir.

> You can use code compiled by the new JVM IR backend only if you've enabled the new backend. Otherwise, you will get an error. Considering this, we don't recommend that library authors switch to the new backend in production.

New modes for generating default methods

When compiling Kotlin code to targets JVM 1.8 and above, you could compile non-abstract methods of Kotlin interfaces into Java's default methods. For this purpose, there was a mechanism that includes the @JvmDefault annotation for marking such methods and the -Xjvm-default compiler option that enables processing of this annotation.

In 1.4.0, we've added a new mode for generating default methods: -Xjvm-default=all compiles all non-abstract methods of Kotlin interfaces to default Java methods. For compatibility with the code that uses the interfaces compiled without default, we also added all-compatibility mode.

For more information about default methods in the Java interop, see the interoperability documentation and this blog post.

Unified exception type for null checks

Starting from Kotlin 1.4.0, all runtime null checks will throw a java.lang.NullPointerException instead of KotlinNullPointerException, IllegalStateException, IllegalArgumentException, and TypeCastException. This applies to: the !! operator, parameter null checks in the method preamble, platform-typed expression null checks, and the as operator with a non-null type. This doesn't apply to lateinit null checks and explicit library function calls like checkNotNull or requireNotNull.

This change increases the number of possible null check optimizations that can be performed either by the Kotlin compiler or by various kinds of bytecode processing tools, such as the Android R8 optimizer.

Note that from a developer's perspective, things won't change that much: the Kotlin code will throw exceptions with the same error messages as before. The type of exception changes, but the information passed stays the same.

Type annotations in the JVM bytecode

Kotlin can now generate type annotations in the JVM bytecode (target version 1.8+), so that they become available in Java reflection at runtime. To emit the type

annotation in the bytecode, follow these steps:

1. Make sure that your declared annotation has a proper annotation target (Java's ElementType.TYPE_USE or Kotlin's AnnotationTarget.TYPE) and retention (AnnotationRetention.RUNTIME).
2. Compile the annotation class declaration to JVM bytecode target version 1.8+. You can specify it with -jvm-target=1.8 compiler option.
3. Compile the code that uses the annotation to JVM bytecode target version 1.8+ (-jvm-target=1.8) and add the -Xemit-jvm-type-annotations compiler option.

Note that the type annotations from the standard library aren't emitted in the bytecode for now because the standard library is compiled with the target version 1.6.

So far, only the basic cases are supported:

- Type annotations on method parameters, method return types and property types;
- Invariant projections of type arguments, such as Smth<@Ann Foo>, Array<@Ann Foo>.

In the following example, the @Foo annotation on the String type can be emitted to the bytecode and then used by the library code:

```
@Target(AnnotationTarget.TYPE)
annotation class Foo

class A {
    fun foo(): @Foo String = "OK"
}
```

Kotlin/JS

On the JS platform, Kotlin 1.4.0 provides the following improvements:

- New Gradle DSL
- New JS IR backend

New Gradle DSL

The kotlin.js Gradle plugin comes with an adjusted Gradle DSL, which provides a number of new configuration options and is more closely aligned to the DSL used by the kotlin-multiplatform plugin. Some of the most impactful changes include:

- Explicit toggles for the creation of executable files via binaries.executable(). Read more about the executing Kotlin/JS and its environment here.
- Configuration of webpack's CSS and style loaders from within the Gradle configuration via cssSupport. Read more about using CSS and style loaders here.
- Improved management for npm dependencies, with mandatory version numbers or semver version ranges, as well as support for development, peer, and optional npm dependencies using devNpm, optionalNpm and peerNpm. Read more about dependency management for npm packages directly from Gradle here.
- Stronger integrations for Dukat, the generator for Kotlin external declarations. External declarations can now be generated at build time, or can be manually generated via a Gradle task.

New JS IR backend

The IR backend for Kotlin/JS, which currently has Alpha stability, provides some new functionality specific to the Kotlin/JS target which is focused around the generated code size through dead code elimination, and improved interoperation with JavaScript and TypeScript, among others.

To enable the Kotlin/JS IR backend, set the key kotlin.js.compiler=ir in your gradle.properties, or pass the IR compiler type to the js function of your Gradle build script:

```
kotlin {
    js(IR) { // or: LEGACY, BOTH
        // ...
    }
    binaries.executable()
}
```

For more detailed information about how to configure the new backend, check out the Kotlin/JS IR compiler documentation.

With the new @JsExport annotation and the ability to generate TypeScript definitions from Kotlin code, the Kotlin/JS IR compiler backend improves JavaScript & TypeScript interoperability. This also makes it easier to integrate Kotlin/JS code with existing tooling, to create hybrid applications and leverage code-sharing functionality in multiplatform projects.

Learn more about the available features in the Kotlin/JS IR compiler backend.

Kotlin/Native

In 1.4.0, Kotlin/Native got a significant number of new features and improvements, including:

- Support for suspending functions in Swift and Objective-C
- Objective-C generics support by default
- Exception handling in Objective-C/Swift interop
- Generate release .dSYMs on Apple targets by default
- Performance improvements
- Simplified management of CocoaPods dependencies

Support for Kotlin's suspending functions in Swift and Objective-C

In 1.4.0, we add the basic support for suspending functions in Swift and Objective-C. Now, when you compile a Kotlin module into an Apple framework, suspending functions are available in it as functions with callbacks (completionHandler in the Swift/Objective-C terminology). When you have such functions in the generated framework's header, you can call them from your Swift or Objective-C code and even override them.

For example, if you write this Kotlin function:

```
suspend fun queryData(id: Int): String = ...
```

...then you can call it from Swift like so:

```
queryData(id: 17) { result, error in
   if let e = error {
       print("ERROR: \(e)")
   } else {
       print(result!)
   }
}
```

Learn more about using suspending functions in Swift and Objective-C.

Objective-C generics support by default

Previous versions of Kotlin provided experimental support for generics in Objective-C interop. Since 1.4.0, Kotlin/Native generates Apple frameworks with generics from Kotlin code by default. In some cases, this may break existing Objective-C or Swift code calling Kotlin frameworks. To have the framework header written without generics, add the -Xno-objc-generics compiler option.

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xno-objc-generics"
        }
    }
}
```

Please note that all specifics and limitations listed in the documentation on interoperability with Objective-C are still valid.

Exception handling in Objective-C/Swift interop

In 1.4.0, we slightly change the Swift API generated from Kotlin with respect to the way exceptions are translated. There is a fundamental difference in error handling between Kotlin and Swift. All Kotlin exceptions are unchecked, while Swift has only checked errors. Thus, to make Swift code aware of expected exceptions, Kotlin functions should be marked with a @Throws annotation specifying a list of potential exception classes.

When compiling to Swift or the Objective-C framework, functions that have or are inheriting @Throws annotation are represented as NSError*-producing methods in Objective-C and as throws methods in Swift.

Previously, any exceptions other than RuntimeException and Error were propagated as NSError. Now this behavior changes: now NSError is thrown only for exceptions that are instances of classes specified as parameters of @Throws annotation (or their subclasses). Other Kotlin exceptions that reach Swift/Objective-C are considered unhandled and cause program termination.

Generate release .dSYMs on Apple targets by default

Starting with 1.4.0, the Kotlin/Native compiler produces debug symbol files (.dSYMs) for release binaries on Darwin platforms by default. This can be disabled with the -Xadd-light-debug=disable compiler option. On other platforms, this option is disabled by default. To toggle this option in Gradle, use:

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

Learn more about crash report symbolication.

Performance improvements

Kotlin/Native has received a number of performance improvements that speed up both the development process and execution. Here are some examples:

- To improve the speed of object allocation, we now offer the mimalloc memory allocator as an alternative to the system allocator. mimalloc works up to two times faster on some benchmarks. Currently, the usage of mimalloc in Kotlin/Native is experimental; you can switch to it using the -Xallocator=mimalloc compiler option.
- We've reworked how C interop libraries are built. With the new tooling, Kotlin/Native produces interop libraries up to 4 times as fast as before, and artifacts are 25% to 30% the size they used to be.
- Overall runtime performance has improved because of optimizations in GC. This improvement will be especially apparent in projects with a large number of long-lived objects. HashMap and HashSet collections now work faster by escaping redundant boxing.
- In 1.3.70 we introduced two new features for improving the performance of Kotlin/Native compilation: caching project dependencies and running the compiler from the Gradle daemon. Since that time, we've managed to fix numerous issues and improve the overall stability of these features.

Simplified management of CocoaPods dependencies

Previously, once you integrated your project with the dependency manager CocoaPods, you could build an iOS, macOS, watchOS, or tvOS part of your project only in Xcode, separate from other parts of your multiplatform project. These other parts could be built in IntelliJ IDEA.

Moreover, every time you added a dependency on an Objective-C library stored in CocoaPods (Pod library), you had to switch from IntelliJ IDEA to Xcode, call pod install, and run the Xcode build there.

Now you can manage Pod dependencies right in IntelliJ IDEA while enjoying the benefits it provides for working with code, such as code highlighting and completion. You can also build the whole Kotlin project with Gradle, without having to switch to Xcode. This means you only have to go to Xcode when you need to write Swift/Objective-C code or run your application on a simulator or device.

Now you can also work with Pod libraries stored locally.

Depending on your needs, you can add dependencies between:

- A Kotlin project and Pod libraries stored remotely in the CocoaPods repository or stored locally on your machine.
- A Kotlin Pod (Kotlin project used as a CocoaPods dependency) and an Xcode project with one or more targets.

Complete the initial configuration, and when you add a new dependency to cocoapods, just re-import the project in IntelliJ IDEA. The new dependency will be added automatically. No additional steps are required.

Learn how to add dependencies.

Kotlin Multiplatform

Support for multiplatform projects is in Alpha. It may change incompatibly and require manual migration in the future. We appreciate your feedback on it in YouTrack.

Kotlin Multiplatform reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming. We continue investing our effort in multiplatform features and improvements:

- Sharing code in several targets with the hierarchical project structure
- Leveraging native libs in the hierarchical structure
- Specifying kotlinx dependencies only once

Multiplatform projects require Gradle 6.0 or later.

Sharing code in several targets with the hierarchical project structure

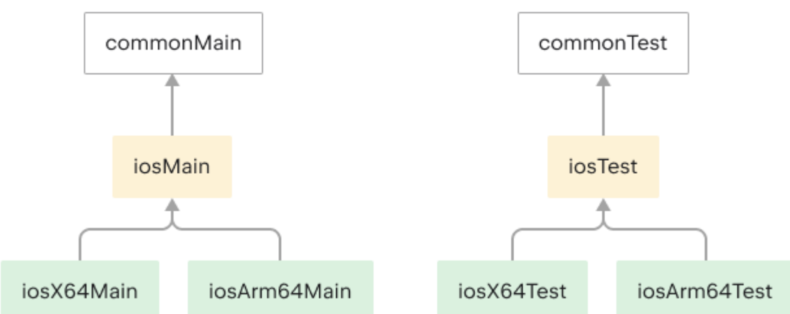
With the new hierarchical project structure support, you can share code among several platforms in a multiplatform project.

Previously, any code added to a multiplatform project could be placed either in a platform-specific source set, which is limited to one target and can't be reused by any other platform, or in a common source set, like commonMain or commonTest, which is shared across all the platforms in the project. In the common source set, you could only call a platform-specific API by using an expect declaration that needs platform-specific actual implementations.

This made it easy to share code on all platforms, but it was not so easy to share between only some of the targets, especially similar ones that could potentially reuse a lot of the common logic and third-party APIs.

For example, in a typical multiplatform project targeting iOS, there are two iOS-related targets: one for iOS ARM64 devices, and the other for the x64 simulator. They have separate platform-specific source sets, but in practice, there is rarely a need for different code for the device and simulator, and their dependencies are much alike. So iOS-specific code could be shared between them.

Apparently, in this setup, it would be desirable to have a shared source set for two iOS targets, with Kotlin/Native code that could still directly call any of the APIs that are common to both the iOS device and the simulator.



Code shared for iOS targets

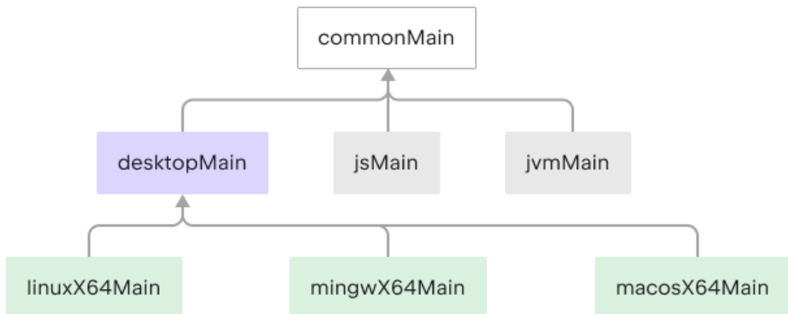
Now you can do this with the hierarchical project structure support, which infers and adapts the API and language features available in each source set based on which targets consume them.

For common combinations of targets, you can create a hierarchical structure with target shortcuts.

For example, create two iOS targets and the shared source set shown above with the ios() shortcut:

```
kotlin {
    ios() // iOS device and simulator targets; iosMain and iosTest source sets
}
```

For other combinations of targets, create a hierarchy manually by connecting the source sets with the dependsOn relation.



Hierarchical structure

Kotlin

```kotlin
kotlin{
    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain)
        }
        val linuxX64Main by getting {
            dependsOn(desktopMain)
        }
        val mingwX64Main by getting {
            dependsOn(desktopMain)
        }
        val macosX64Main by getting {
            dependsOn(desktopMain)
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        desktopMain {
            dependsOn(commonMain)
        }
        linuxX64Main {
            dependsOn(desktopMain)
        }
        mingwX64Main {
            dependsOn(desktopMain)
        }
        macosX64Main {
            dependsOn(desktopMain)
        }
    }
}
```

Thanks to the hierarchical project structure, libraries can also provide common APIs for a subset of targets. Learn more about sharing code in libraries.

Leveraging native libs in the hierarchical structure

You can use platform-dependent libraries, such as Foundation, UIKit, and POSIX, in source sets shared among several native targets. This can help you share more native code without being limited by platform-specific dependencies.

No additional steps are required – everything is done automatically. IntelliJ IDEA will help you detect common declarations that you can use in the shared code.

Learn more about usage of platform-dependent libraries.

Specifying dependencies only once

From now on, instead of specifying dependencies on different variants of the same library in shared and platform-specific source sets where it is used, you should specify a dependency only once in the shared source set.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1'
            }
        }
    }
}
```

Don't use kotlinx library artifact names with suffixes specifying the platform, such as -common, -native, or similar, as they are NOT supported anymore. Instead, use the library base artifact name, which in the example above is kotlinx-coroutines-core.

However, the change doesn't currently affect:

- The stdlib library – starting from Kotlin 1.4.0, the stdlib dependency is added automatically.
- The kotlin.test library – you should still use test-common and test-annotations-common. These dependencies will be addressed later.

If you need a dependency only for a specific platform, you can still use platform-specific variants of standard and kotlinx libraries with such suffixes as -jvm or-js, for example kotlinx-coroutines-core-jvm.

Learn more about configuring dependencies.

Gradle project improvements

Besides Gradle project features and improvements that are specific to Kotlin Multiplatform, Kotlin/JVM, Kotlin/Native, and Kotlin/JS, there are several changes applicable to all Kotlin Gradle projects:

- Dependency on the standard library is now added by default
- Kotlin projects require a recent version of Gradle
- Improved support for Kotlin Gradle DSL in the IDE

Dependency on the standard library added by default

You no longer need to declare a dependency on the stdlib library in any Kotlin Gradle project, including a multiplatform one. The dependency is added by default.

The automatically added standard library will be the same version of the Kotlin Gradle plugin, since they have the same versioning.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin will select the appropriate JVM standard library depending on the kotlinOptions.jvmTarget compiler option of your Gradle build script.

Learn how to change the default behavior.

Minimum Gradle version for Kotlin projects

To enjoy the new features in your Kotlin projects, update Gradle to the latest version. Multiplatform projects require Gradle 6.0 or later, while other Kotlin projects

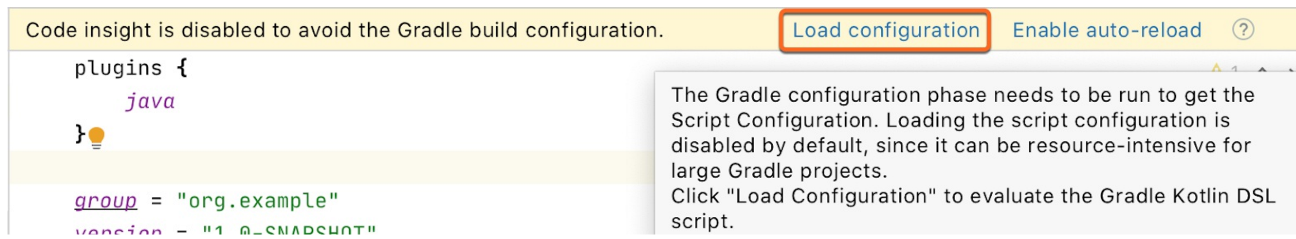
work with Gradle 5.4 or later.

Improved *.gradle.kts support in the IDE

In 1.4.0, we continued improving the IDE support for Gradle Kotlin DSL scripts (*.gradle.kts files). Here is what the new version brings:

- Explicit loading of script configurations for better performance. Previously, the changes you make to the build script were loaded automatically in the background. To improve the performance, we've disabled the automatic loading of build script configuration in 1.4.0. Now the IDE loads the changes only when you explicitly apply them.

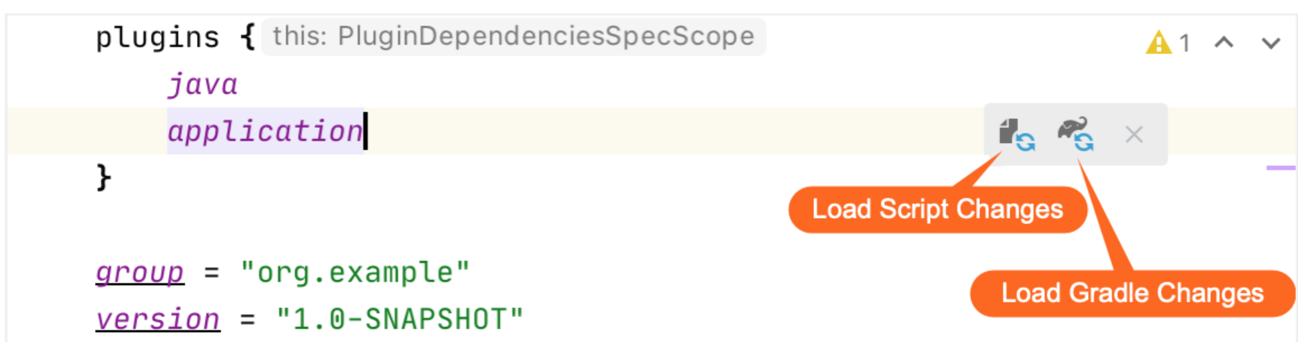
  In Gradle versions earlier than 6.0, you need to manually load the script configuration by clicking Load Configuration in the editor.



*.gradle.kts – Load Configuration

  In Gradle 6.0 and above, you can explicitly apply changes by clicking Load Gradle Changes or by reimporting the Gradle project.

  We've added one more action in IntelliJ IDEA 2020.1 with Gradle 6.0 and above – Load Script Configurations, which loads changes to the script configurations without updating the whole project. This takes much less time than reimporting the whole project.

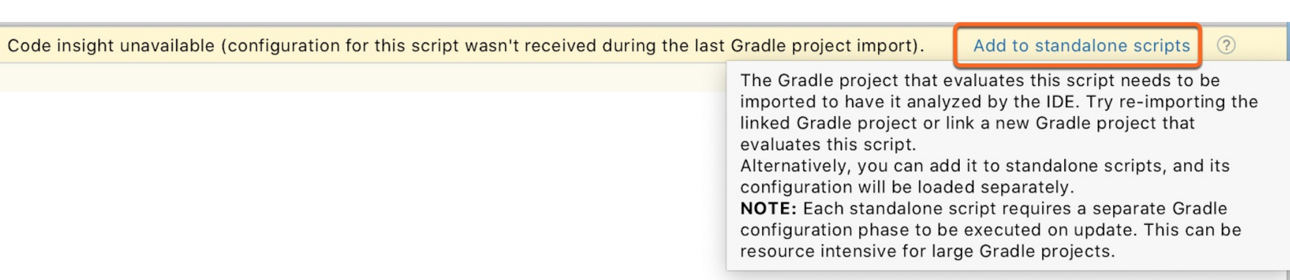


*.gradle.kts – Load Script Changes and Load Gradle Changes

  You should also Load Script Configurations for newly created scripts or when you open a project with new Kotlin plugin for the first time.

  With Gradle 6.0 and above, you are now able to load all scripts at once as opposed to the previous implementation where they were loaded individually. Since each request requires the Gradle configuration phase to be executed, this could be resource-intensive for large Gradle projects.

  Currently, such loading is limited to build.gradle.kts and settings.gradle.kts files (please vote for the related issue). To enable highlighting for init.gradle.kts or applied script plugins, use the old mechanism – adding them to standalone scripts. Configuration for that scripts will be loaded separately when you need it. You can also enable auto-reload for such scripts.



*.gradle.kts – Add to standalone scripts

- Better error reporting. Previously you could only see errors from the Gradle Daemon in separate log files. Now the Gradle Daemon returns all the information about errors directly and shows it in the Build tool window. This saves you both time and effort.

Standard library

Here is the list of the most significant changes to the Kotlin standard library in 1.4.0:

- Common exception processing API
- New functions for arrays and collections
- Functions for string manipulations
- Bit operations
- Delegated properties improvements
- Converting from KType to Java Type
- Proguard configurations for Kotlin reflection
- Improving the existing API
- module-info descriptors for stdlib artifacts
- Deprecations
- Exclusion of the deprecated experimental coroutines

Common exception processing API

The following API elements have been moved to the common library:

- Throwable.stackTraceToString() extension function, which returns the detailed description of this throwable with its stack trace, and Throwable.printStackTrace(), which prints this description to the standard error output.
- Throwable.addSuppressed() function, which lets you specify the exceptions that were suppressed in order to deliver the exception, and the Throwable.suppressedExceptions property, which returns a list of all the suppressed exceptions.
- @Throws annotation, which lists exception types that will be checked when the function is compiled to a platform method (on JVM or native platforms).

New functions for arrays and collections

Collections

In 1.4.0, the standard library includes a number of useful functions for working with collections:

- setOfNotNull(), which makes a set consisting of all the non-null items among the provided arguments.

```
fun main() {
//sampleStart
    val set = setOfNotNull(null, 1, 2, 0, null)
    println(set)
//sampleEnd
}
```

- shuffled() for sequences.

```
fun main() {
//sampleStart
    val numbers = (0 until 50).asSequence()
    val result = numbers.map { it * 2 }.shuffled().take(5)
    println(result.toList()) //five random even numbers below 100
//sampleEnd
}
```

- *Indexed() counterparts for onEach() and flatMap(). The operation that they apply to the collection elements has the element index as a parameter.

```
fun main() {
//sampleStart
    listOf("a", "b", "c", "d").onEachIndexed {
        index, item -> println(index.toString() + ":" + item)
    }

   val list = listOf("hello", "kot", "lin", "world")
          val kotlin = list.flatMapIndexed { index, item ->
              if (index in 1..2) item.toList() else emptyList()
          }
//sampleEnd
          println(kotlin)
}
```

- *OrNull() counterparts randomOrNull(), reduceOrNull(), and reduceIndexedOrNull(). They return null on empty collections.

```
fun main() {
//sampleStart
     val empty = emptyList<Int>()
     empty.reduceOrNull { a, b -> a + b }
     //empty.reduce { a, b -> a + b } // Exception: Empty collection can't be reduced.
//sampleEnd
}
```

- runningFold(), its synonym scan(), and runningReduce() apply the given operation to the collection elements sequentially, similarly tofold() and reduce(); the difference is that these new functions return the whole sequence of intermediate results.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(0, 1, 2, 3, 4, 5)
    val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }
    val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }
//sampleEnd
    println(runningReduceSum.toString())
    println(runningFoldSum.toString())
}
```

- sumOf() takes a selector function and returns a sum of its values for all elements of a collection. sumOf() can produce sums of the types Int, Long, Double, UInt, and ULong. On the JVM, BigInteger and BigDecimal are also available.

```
data class OrderItem(val name: String, val price: Double, val count: Int)

fun main() {
//sampleStart
    val order = listOf<OrderItem>(
        OrderItem("Cake", price = 10.0, count = 1),
        OrderItem("Coffee", price = 2.5, count = 3),
        OrderItem("Tea", price = 1.5, count = 2))

    val total = order.sumOf { it.price * it.count } // Double
    val count = order.sumOf { it.count } // Int
//sampleEnd
    println("You've ordered $count items that cost $total in total")
}
```

- The min() and max() functions have been renamed to minOrNull() and maxOrNull() to comply with the naming convention used across the Kotlin collections API. An *OrNull suffix in the function name means that it returns null if the receiver collection is empty. The same applies to minBy(), maxBy(), minWith(), maxWith() – in 1.4, they have *OrNull() synonyms.

- The new minOf() and maxOf() extension functions return the minimum and the maximum value of the given selector function on the collection items.

```
data class OrderItem(val name: String, val price: Double, val count: Int)

fun main() {
//sampleStart
    val order = listOf<OrderItem>(
        OrderItem("Cake", price = 10.0, count = 1),
        OrderItem("Coffee", price = 2.5, count = 3),
        OrderItem("Tea", price = 1.5, count = 2))
    val highestPrice = order.maxOf { it.price }
//sampleEnd
    println("The most expensive item in the order costs $highestPrice")
```

```
}
```

There are also minOfWith() and maxOfWith(), which take a Comparator as an argument, and *OrNull() versions of all four functions that return null on empty collections.

- New overloads for flatMap and flatMapTo let you use transformations with return types that don't match the receiver type, namely:
  - Transformations to Sequence on Iterable, Array, and Map
  - Transformations to Iterable on Sequence

```
fun main() {
//sampleStart
    val list = listOf("kot", "lin")
    val lettersList = list.flatMap { it.asSequence() }
    val lettersSeq = list.asSequence().flatMap { it.toList() }
//sampleEnd
    println(lettersList)
    println(lettersSeq.toList())
}
```

- removeFirst() and removeLast() shortcuts for removing elements from mutable lists, and *orNull() counterparts of these functions.

Arrays

To provide a consistent experience when working with different container types, we've also added new functions for arrays:

- shuffle() puts the array elements in a random order.
- onEach() performs the given action on each array element and returns the array itself.
- associateWith() and associateWithTo() build maps with the array elements as keys.
- reverse() for array subranges reverses the order of the elements in the subrange.
- sortDescending() for array subranges sorts the elements in the subrange in descending order.
- sort() and sortWith() for array subranges are now available in the common library.

```
fun main() {
//sampleStart
    var language = ""
    val letters = arrayOf("k", "o", "t", "l", "i", "n")
    val fileExt = letters.onEach { language += it }
       .filterNot { it in "aeuio" }.take(2)
       .joinToString(prefix = ".", separator = "")
    println(language) // "kotlin"
    println(fileExt) // ".kt"

    letters.shuffle()
    letters.reverse(0, 3)
    letters.sortDescending(2, 5)
    println(letters.contentToString()) // [k, o, t, l, i, n]
//sampleEnd
}
```

Additionally, there are new functions for conversions between CharArray/ByteArray and String:

- ByteArray.decodeToString() and String.encodeToByteArray()
- CharArray.concatToString() and String.toCharArray()

```
fun main() {
//sampleStart
 val str = "kotlin"
    val array = str.toCharArray()
    println(array.concatToString())
//sampleEnd
}
```

ArrayDeque

We've also added the ArrayDeque class – an implementation of a double-ended queue. A double-ended queue lets you add or remove elements both at the beginning or end of the queue in an amortized constant time. You can use a double-ended queue by default when you need a queue or a stack in your code.

```
fun main() {
    val deque = ArrayDeque(listOf(1, 2, 3))

    deque.addFirst(0)
    deque.addLast(4)
    println(deque) // [0, 1, 2, 3, 4]

    println(deque.first()) // 0
    println(deque.last()) // 4

    deque.removeFirst()
    deque.removeLast()
    println(deque) // [1, 2, 3]
}
```

The ArrayDeque implementation uses a resizable array underneath: it stores the contents in a circular buffer, an Array, and resizes this Array only when it becomes full.

Functions for string manipulations

The standard library in 1.4.0 includes a number of improvements in the API for string manipulation:

- StringBuilder has useful new extension functions: set(), setRange(), deleteAt(), deleteRange(), appendRange(), and others.

```
    fun main() {
    //sampleStart
        val sb = StringBuilder("Bye Kotlin 1.3.72")
        sb.deleteRange(0, 3)
        sb.insertRange(0, "Hello", 0 ,5)
        sb.set(15, '4')
        sb.setRange(17, 19, "0")
        print(sb.toString())
    //sampleEnd
    }
```

- Some existing functions of StringBuilder are available in the common library. Among them are append(), insert(), substring(), setLength(), and more.
- New functions Appendable.appendLine() and StringBuilder.appendLine() have been added to the common library. They replace the JVM-only appendln() functions of these classes.

```
fun main() {
//sampleStart
    println(buildString {
        appendLine("Hello,")
        appendLine("world")
    })
//sampleEnd
}
```

Bit operations

New functions for bit manipulations:

- countOneBits()
- countLeadingZeroBits()
- countTrailingZeroBits()
- takeHighestOneBit()
- takeLowestOneBit()
- rotateLeft() and rotateRight() (experimental)

```
fun main() {
```

```
//sampleStart
    val number = "1010000".toInt(radix = 2)
    println(number.countOneBits())
    println(number.countTrailingZeroBits())
    println(number.takeHighestOneBit().toString(2))
//sampleEnd
}
```

Delegated properties improvements

In 1.4.0, we have added new features to improve your experience with delegated properties in Kotlin:

- Now a property can be delegated to another property.
- A new interface PropertyDelegateProvider helps create delegate providers in a single declaration.
- ReadWriteProperty now extends ReadOnlyProperty so you can use both of them for read-only properties.

Aside from the new API, we've made some optimizations that reduce the resulting bytecode size. These optimizations are described in this blog post.

Learn more about delegated properties.

Converting from KType to Java Type

A new extension property KType.javaType (currently experimental) in the stdlib helps you obtain a java.lang.reflect.Type from a Kotlin type without using the whole kotlin-reflect dependency.

```
import kotlin.reflect.javaType
import kotlin.reflect.typeOf

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T> accessReifiedTypeArg() {
   val kType = typeOf<T>()
   println("Kotlin type: $kType")
   println("Java type: ${kType.javaType}")
}

@OptIn(ExperimentalStdlibApi::class)
fun main() {
   accessReifiedTypeArg<String>()
   // Kotlin type: kotlin.String
   // Java type: class java.lang.String

   accessReifiedTypeArg<List<String>>()
   // Kotlin type: kotlin.collections.List<kotlin.String>
   // Java type: java.util.List<java.lang.String>
}
```

Proguard configurations for Kotlin reflection

Starting from 1.4.0, we have embedded Proguard/R8 configurations for Kotlin Reflection in kotlin-reflect.jar. With this in place, most Android projects using R8 or Proguard should work with kotlin-reflect without needing any additional configuration. You no longer need to copy-paste the Proguard rules for kotlin-reflect internals. But note that you still need to explicitly list all the APIs you're going to reflect on.

Improving the existing API

- Several functions now work on null receivers, for example:
  - toBoolean() on strings
  - contentEquals(), contentHashcode(), contentToString() on arrays
- NaN, NEGATIVE_INFINITY, and POSITIVE_INFINITY in Double and Float are now defined as const, so you can use them as annotation arguments.
- New constants SIZE_BITS and SIZE_BYTES in Double and Float contain the number of bits and bytes used to represent an instance of the type in binary form.
- The maxOf() and minOf() top-level functions can accept a variable number of arguments (vararg).

module-info descriptors for stdlib artifacts

Kotlin 1.4.0 adds module-info.java module information to default standard library artifacts. This lets you use them with jlink tool, which generates custom Java runtime images containing only the platform modules that are required for your app. You could already use jlink with Kotlin standard library artifacts, but you had to use separate artifacts to do so – the ones with the "modular" classifier – and the whole setup wasn't straightforward.
In Android, make sure you use the Android Gradle plugin version 3.2 or higher, which can correctly process jar files with module-info.

Deprecations

toShort() and toByte() of Double and Float

We've deprecated the functions toShort() and toByte() on Double and Float because they could lead to unexpected results because of the narrow value range and smaller variable size.

To convert floating-point numbers to Byte or Short, use the two-step conversion: first, convert them to Int, and then convert them again to the target type.

contains(), indexOf(), and lastIndexOf() on floating-point arrays

We've deprecated the contains(), indexOf(), and lastIndexOf() extension functions of FloatArray and DoubleArray because they use the IEEE 754 standard equality, which contradicts the total order equality in some corner cases. See this issue for details.

min() and max() collection functions

We've deprecated the min() and max() collection functions in favor of minOrNull() and maxOrNull(), which more properly reflect their behavior – returning null on empty collections. See this issue for details.

Exclusion of the deprecated experimental coroutines

The kotlin.coroutines.experimental API was deprecated in favor of kotlin.coroutines in 1.3.0. In 1.4.0, we're completing the deprecation cycle for kotlin.coroutines.experimental by removing it from the standard library. For those who still use it on the JVM, we've provided a compatibility artifact kotlin-coroutines-experimental-compat.jar with all the experimental coroutines APIs. We've published it to Maven, and we include it in the Kotlin distribution alongside the standard library.

Stable JSON serialization

With Kotlin 1.4.0, we are shipping the first stable version of kotlinx.serialization - 1.0.0-RC. Now we are pleased to declare the JSON serialization API in kotlinx-serialization-core (previously known as kotlinx-serialization-runtime) stable. Libraries for other serialization formats remain experimental, along with some advanced parts of the core library.

We have significantly reworked the API for JSON serialization to make it more consistent and easier to use. From now on, we'll continue developing the JSON serialization API in a backward-compatible manner. However, if you have used previous versions of it, you'll need to rewrite some of your code when migrating to 1.0.0-RC. To help you with this, we also offer the Kotlin Serialization Guide – the complete set of documentation for kotlinx.serialization. It will guide you through the process of using the most important features and it can help you address any issues that you might face.

> Note: kotlinx-serialization 1.0.0-RC only works with Kotlin compiler 1.4. Earlier compiler versions are not compatible.

Scripting and REPL

In 1.4.0, scripting in Kotlin benefits from a number of functional and performance improvements along with other updates. Here are some of the key changes:

- New dependencies resolution API
- New REPL API
- Compiled scripts cache
- Artifacts renaming

To help you become more familiar with scripting in Kotlin, we've prepared a project with examples. It contains examples of the standard scripts (*.main.kts) and examples of uses of the Kotlin Scripting API and custom script definitions. Please give it a try and share your feedback using our issue tracker.

New dependencies resolution API

In 1.4.0, we've introduced a new API for resolving external dependencies (such as Maven artifacts), along with implementations for it. This API is published in the new artifacts kotlin-scripting-dependencies and kotlin-scripting-dependencies-maven. The previous dependency resolution functionality in kotlin-script-util library is now deprecated.

New REPL API

The new experimental REPL API is now a part of the Kotlin Scripting API. There are also several implementations of it in the published artifacts, and some have advanced functionality, such as code completion. We use this API in the Kotlin Jupyter kernel and now you can try it in your own custom shells and REPLs.

Compiled scripts cache

The Kotlin Scripting API now provides the ability to implement a compiled scripts cache, significantly speeding up subsequent executions of unchanged scripts. Our default advanced script implementation kotlin-main-kts already has its own cache.

Artifacts renaming

In order to avoid confusion about artifact names, we've renamed kotlin-scripting-jsr223-embeddable and kotlin-scripting-jvm-host-embeddable to just kotlin-scripting-jsr223 and kotlin-scripting-jvm-host. These artifacts depend on the kotlin-compiler-embeddable artifact, which shades the bundled third-party libraries to avoid usage conflicts. With this renaming, we're making the usage of kotlin-compiler-embeddable (which is safer in general) the default for scripting artifacts. If, for some reason, you need artifacts that depend on the unshaded kotlin-compiler, use the artifact versions with the -unshaded suffix, such as kotlin-scripting-jsr223-unshaded. Note that this renaming affects only the scripting artifacts that are supposed to be used directly; names of other artifacts remain unchanged.

Migrating to Kotlin 1.4.0

The Kotlin plugin's migration tools help you migrate your projects from earlier versions of Kotlin to 1.4.0.

Just change the Kotlin version to 1.4.0 and re-import your Gradle or Maven project. The IDE will then ask you about migration.

If you agree, it will run migration code inspections that will check your code and suggest corrections for anything that doesn't work or that is not recommended in 1.4.0.

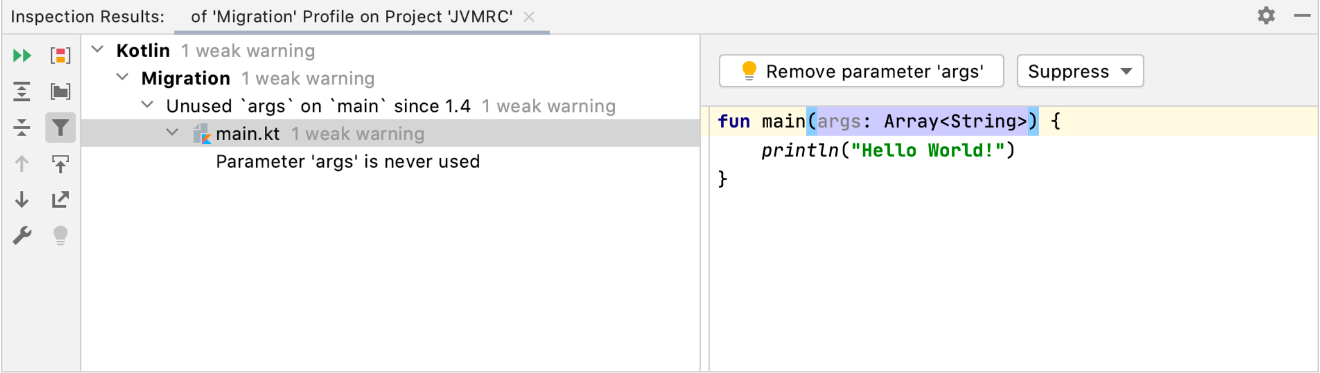
Kotlin Migration

Migrations for Kotlin code are available...

Run migrations

Run migration

Code inspections have different severity levels, to help you decide which suggestions to accept and which to ignore.



Migration inspections

Kotlin 1.4.0 is a feature release and therefore can bring incompatible changes to the language. Find the detailed list of such changes in the Compatibility Guide for Kotlin 1.4.

What's new in Kotlin 1.3

Released: 29 October 2018

Coroutines release

After some long and extensive battle testing, coroutines are now released! It means that from Kotlin 1.3 the language support and the API are fully stable. Check out the new coroutines overview page.

Kotlin 1.3 introduces callable references on suspend-functions and support of coroutines in the reflection API.

Kotlin/Native

Kotlin 1.3 continues to improve and polish the Native target. See the Kotlin/Native overview for details.

Multiplatform projects

In 1.3, we've completely reworked the model of multiplatform projects in order to improve expressiveness and flexibility, and to make sharing common code easier. Also, Kotlin/Native is now supported as one of the targets!

The key differences to the old model are:

- In the old model, common and platform-specific code needed to be placed in separate modules, linked by expectedBy dependencies. Now, common and platform-specific code is placed in different source roots of the same module, making projects easier to configure.
- There is now a large number of preset platform configurations for different supported platforms.
- The dependencies configuration has been changed; dependencies are now specified separately for each source root.
- Source sets can now be shared between an arbitrary subset of platforms (for example, in a module that targets JS, Android and iOS, you can have a source set that is shared only between Android and iOS).
- Publishing multiplatform libraries is now supported.

For more information, please refer to the multiplatform programming documentation.

Contracts

The Kotlin compiler does extensive static analysis to provide warnings and reduce boilerplate. One of the most notable features is smartcasts — with the ability to perform a cast automatically based on the performed type checks:

```
fun foo(s: String?) {
    if (s != null) s.length // Compiler automatically casts 's' to 'String'
}
```

However, as soon as these checks are extracted in a separate function, all the smartcasts immediately disappear:

```
fun String?.isNotNull(): Boolean = this != null

fun foo(s: String?) {
    if (s.isNotNull()) s.length // No smartcast :(
}
```

To improve the behavior in such cases, Kotlin 1.3 introduces experimental mechanism called contracts.

Contracts allow a function to explicitly describe its behavior in a way which is understood by the compiler. Currently, two wide classes of cases are supported:

- Improving smartcasts analysis by declaring the relation between a function's call outcome and the passed arguments values:

```
fun require(condition: Boolean) {
    // This is a syntax form which tells the compiler:
    // "if this function returns successfully, then the passed 'condition' is true"
    contract { returns() implies condition }
    if (!condition) throw IllegalArgumentException(...)
}

fun foo(s: String?) {
    require(s is String)
    // s is smartcast to 'String' here, because otherwise
    // 'require' would have thrown an exception
}
```

- Improving the variable initialization analysis in the presence of high-order functions:

```
fun synchronize(lock: Any?, block: () -> Unit) {
    // It tells the compiler:
    // "This function will invoke 'block' here and now, and exactly one time"
    contract { callsInPlace(block, EXACTLY_ONCE) }
}

fun foo() {
    val x: Int
    synchronize(lock) {
        x = 42 // Compiler knows that lambda passed to 'synchronize' is called
               // exactly once, so no reassignment is reported
    }
    println(x) // Compiler knows that lambda will be definitely called, performing
               // initialization, so 'x' is considered to be initialized here
}
```

Contracts in stdlib

stdlib already makes use of contracts, which leads to improvements in the analyses described above. This part of contracts is stable, meaning that you can benefit from the improved analysis right now without any additional opt-ins:

```
//sampleStart
fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // Yay, smartcast to not-null!
    }
}
//sampleEnd
fun main() {
    bar(null)
    bar("42")
}
```

Custom contracts

It is possible to declare contracts for your own functions, but this feature is experimental, as the current syntax is in a state of early prototype and will most probably be changed. Also please note that currently the Kotlin compiler does not verify contracts, so it's the responsibility of the programmer to write correct and sound contracts.

Custom contracts are introduced by a call to contract stdlib function, which provides DSL scope:

```
fun String?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }
    return this == null || isEmpty()
}
```

See the details on the syntax as well as the compatibility notice in the KEEP.

Capturing when subject in a variable

In Kotlin 1.3, it is now possible to capture the when subject into a variable:

```
fun Request.getBody() =
        when (val response = executeRequest()) {
            is Success -> response.body
            is HttpError -> throw HttpException(response.status)
        }
```

While it was already possible to extract this variable just before when, val in when has its scope properly restricted to the body of when, and so preventing namespace pollution. See the full documentation on when here.

@JvmStatic and @JvmField in companions of interfaces

With Kotlin 1.3, it is possible to mark members of a companion object of interfaces with annotations @JvmStatic and @JvmField. In the classfile, such members will be lifted to the corresponding interface and marked as static.

For example, the following Kotlin code:

```
interface Foo {
    companion object {
        @JvmField
        val answer: Int = 42

        @JvmStatic
        fun sayHello() {
            println("Hello, world!")
        }
    }
}
```

It is equivalent to this Java code:

```
interface Foo {
    public static int answer = 42;
    public static void sayHello() {
        // ...
    }
}
```

Nested declarations in annotation classes

In Kotlin 1.3, it is possible for annotations to have nested classes, interfaces, objects, and companions:

```
annotation class Foo {
    enum class Direction { UP, DOWN, LEFT, RIGHT }

    annotation class Bar

    companion object {
        fun foo(): Int = 42
        val bar: Int = 42
    }
}
```

Parameterless main

By convention, the entry point of a Kotlin program is a function with a signature like main(args: Array<String>), where args represent the command-line arguments passed to the program. However, not every application supports command-line arguments, so this parameter often ends up not being used.

Kotlin 1.3 introduced a simpler form of main which takes no parameters. Now "Hello, World" in Kotlin is 19 characters shorter!

```
fun main() {
    println("Hello, world!")
}
```

Functions with big arity

In Kotlin, functional types are represented as generic classes taking a different number of parameters: Function0<R>, Function1<P0, R>, Function2<P0, P1, R>, ... This approach has a problem in that this list is finite, and it currently ends with Function22.

Kotlin 1.3 relaxes this limitation and adds support for functions with bigger arity:

```
fun trueEnterpriseComesToKotlin(block: (Any, Any, ... /* 42 more */, Any) -> Any) {
    block(Any(), Any(), ..., Any())
}
```

Progressive mode

Kotlin cares a lot about stability and backward compatibility of code: Kotlin compatibility policy says that breaking changes (e.g., a change which makes the code that used to compile fine, not compile anymore) can be introduced only in the major releases (1.2, 1.3, etc.).

We believe that a lot of users could use a much faster cycle where critical compiler bug fixes arrive immediately, making the code more safe and correct. So, Kotlin 1.3 introduces the progressive compiler mode, which can be enabled by passing the argument -progressive to the compiler.

In the progressive mode, some fixes in language semantics can arrive immediately. All these fixes have two important properties:

- They preserve backward compatibility of source code with older compilers, meaning that all the code which is compilable by the progressive compiler will be compiled fine by non-progressive one.
- They only make code safer in some sense — e.g., some unsound smartcast can be forbidden, behavior of the generated code may be changed to be more predictable/stable, and so on.

Enabling the progressive mode can require you to rewrite some of your code, but it shouldn't be too much — all the fixes enabled under progressive are carefully handpicked, reviewed, and provided with tooling migration assistance. We expect that the progressive mode will be a nice choice for any actively maintained codebases which are updated to the latest language versions quickly.

Inline classes

> Inline classes are in Alpha. They may change incompatibly and require manual migration in the future. We appreciate your feedback on it in YouTrack. See details in the reference.

Kotlin 1.3 introduces a new kind of declaration — inline class. Inline classes can be viewed as a restricted version of the usual classes, in particular, inline classes must have exactly one property:

```
inline class Name(val s: String)
```

The Kotlin compiler will use this restriction to aggressively optimize runtime representation of inline classes and substitute their instances with the value of the underlying property where possible removing constructor calls, GC pressure, and enabling other optimizations:

```
inline class Name(val s: String)
//sampleStart
fun main() {
    // In the next line no constructor call happens, and
    // at the runtime 'name' contains just string "Kotlin"
    val name = Name("Kotlin")
    println(name.s)
}
//sampleEnd
```

See reference for inline classes for details.

Unsigned integers

> Unsigned integers are in Beta. Their implementation is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you will have to make.

Kotlin 1.3 introduces unsigned integer types:

- kotlin.UByte: an unsigned 8-bit integer, ranges from 0 to 255
- kotlin.UShort: an unsigned 16-bit integer, ranges from 0 to 65535
- kotlin.UInt: an unsigned 32-bit integer, ranges from 0 to 2^32 - 1
- kotlin.ULong: an unsigned 64-bit integer, ranges from 0 to 2^64 - 1

Most of the functionality of signed types are supported for unsigned counterparts too:

```
fun main() {
//sampleStart
// You can define unsigned types using literal suffixes
val uint = 42u
val ulong = 42uL
val ubyte: UByte = 255u

// You can convert signed types to unsigned and vice versa via stdlib extensions:
val int = uint.toInt()
val byte = ubyte.toByte()
val ulong2 = byte.toULong()

// Unsigned types support similar operators:
val x = 20u + 22u
val y = 1u shl 8
val z = "128".toUByte()
val range = 1u..5u
//sampleEnd
println("ubyte: $ubyte, byte: $byte, ulong2: $ulong2")
println("x: $x, y: $y, z: $z, range: $range")
}
```

See reference for details.

@JvmDefault

> @JvmDefault is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin targets a wide range of the Java versions, including Java 6 and Java 7, where default methods in the interfaces are not allowed. For your convenience, the Kotlin compiler works around that limitation, but this workaround isn't compatible with the default methods, introduced in Java 8.

This could be an issue for Java-interoperability, so Kotlin 1.3 introduces the @JvmDefault annotation. Methods annotated with this annotation will be generated as default methods for JVM:

```
interface Foo {
    // Will be generated as 'default' method
    @JvmDefault
    fun foo(): Int = 42
}
```

> Warning! Annotating your API with @JvmDefault has serious implications on binary compatibility. Make sure to carefully read the reference page before using @JvmDefault in production.

Standard library

Multiplatform random

Prior to Kotlin 1.3, there was no uniform way to generate random numbers on all platforms — we had to resort to platform-specific solutions like java.util.Random on JVM. This release fixes this issue by introducing the class kotlin.random.Random, which is available on all platforms:

```
import kotlin.random.Random

fun main() {
//sampleStart
    val number = Random.nextInt(42)  // number is in range [0, limit)
    println(number)
//sampleEnd
}
```

isNullOrEmpty and orEmpty extensions

isNullOrEmpty and orEmpty extensions for some types are already present in stdlib. The first one returns true if the receiver is null or empty, and the second one falls back to an empty instance if the receiver is null. Kotlin 1.3 provides similar extensions on collections, maps, and arrays of objects.

Copy elements between two existing arrays

The array.copyInto(targetArray, targetOffset, startIndex, endIndex) functions for the existing array types, including the unsigned arrays, make it easier to implement array-based containers in pure Kotlin.

```
fun main() {
//sampleStart
    val sourceArr = arrayOf("k", "o", "t", "l", "i", "n")
    val targetArr = sourceArr.copyInto(arrayOfNulls<String>(6), 3, startIndex = 3, endIndex = 6)
    println(targetArr.contentToString())

    sourceArr.copyInto(targetArr, startIndex = 0, endIndex = 3)
    println(targetArr.contentToString())
//sampleEnd
}
```

associateWith

It is quite a common situation to have a list of keys and want to build a map by associating each of these keys with some value. It was possible to do it before with the associate { it to getValue(it) } function, but now we're introducing a more efficient and easy to explore alternative: keys.associateWith { getValue(it) }.

```
fun main() {
//sampleStart
    val keys = 'a'..'f'
    val map = keys.associateWith { it.toString().repeat(5).capitalize() }
    map.forEach { println(it) }
//sampleEnd
}
```

ifEmpty and ifBlank functions

Collections, maps, object arrays, char sequences, and sequences now have an ifEmpty function, which allows specifying a fallback value that will be used instead of the receiver if it is empty:

```
fun main() {
//sampleStart
    fun printAllUppercase(data: List<String>) {
        val result = data
        .filter { it.all { c -> c.isUpperCase() } }
            .ifEmpty { listOf("<no uppercase>") }
        result.forEach { println(it) }
    }

    printAllUppercase(listOf("foo", "Bar"))
    printAllUppercase(listOf("FOO", "BAR"))
//sampleEnd
}
```

Char sequences and strings in addition have an ifBlank extension that does the same thing as ifEmpty but checks for a string being all whitespace instead of

empty.

```kotlin
fun main() {
//sampleStart
    val s = "    \n"
    println(s.ifBlank { "<blank>" })
    println(s.ifBlank { null })
//sampleEnd
}
```

Sealed classes in reflection

We've added a new API to kotlin-reflect that can be used to enumerate all the direct subtypes of a sealed class, namely KClass.sealedSubclasses.

Smaller changes

- Boolean type now has companion.
- Any?.hashCode() extension that returns 0 for null.
- Char now provides MIN_VALUE and MAX_VALUE constants.
- SIZE_BYTES and SIZE_BITS constants in primitive type companions.

Tooling

Code style support in IDE

Kotlin 1.3 introduces support for the recommended code style in IntelliJ IDEA. Check out this page for the migration guidelines.

kotlinx.serialization

kotlinx.serialization is a library which provides multiplatform support for (de)serializing objects in Kotlin. Previously, it was a separate project, but since Kotlin 1.3, it ships with the Kotlin compiler distribution on par with the other compiler plugins. The main difference is that you don't need to manually watch out for the Serialization IDE Plugin being compatible with the Kotlin IDE plugin version you're using: now the Kotlin IDE plugin already includes serialization!

See here for details.

> Even though kotlinx.serialization now ships with the Kotlin Compiler distribution, it is still considered to be an experimental feature in Kotlin 1.3.

Scripting update

> Scripting is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin 1.3 continues to evolve and improve scripting API, introducing some experimental support for scripts customization, such as adding external properties, providing static or dynamic dependencies, and so on.

For additional details, please consult the KEEP-75.

Scratches support

Kotlin 1.3 introduces support for runnable Kotlin scratch files. Scratch file is a kotlin script file with the .kts extension that you can run and get evaluation results directly in the editor.

Consult the general Scratches documentation for details.

What's new in Kotlin 1.2

Released: 28 November 2017

Table of contents

- Multiplatform projects
- Other language features
- Standard library
- JVM backend
- JavaScript backend

Multiplatform projects (experimental)

Multiplatform projects are a new experimental feature in Kotlin 1.2, allowing you to reuse code between target platforms supported by Kotlin – JVM, JavaScript, and (in the future) Native. In a multiplatform project, you have three kinds of modules:

- A common module contains code that is not specific to any platform, as well as declarations without implementation of platform-dependent APIs.
- A platform module contains implementations of platform-dependent declarations in the common module for a specific platform, as well as other platform-dependent code.
- A regular module targets a specific platform and can either be a dependency of platform modules or depend on platform modules.

When you compile a multiplatform project for a specific platform, the code for both the common and platform-specific parts is generated.

A key feature of the multiplatform project support is the possibility to express dependencies of common code on platform-specific parts through expected and actual declarations. An expected declaration specifies an API (class, interface, annotation, top-level declaration etc.). An actual declaration is either a platform-dependent implementation of the API or a type alias referring to an existing implementation of the API in an external library. Here's an example:

In the common code:

```
// expected platform-specific API:
expect fun hello(world: String): String

fun greet() {
    // usage of the expected API:
    val greeting = hello("multiplatform world")
    println(greeting)
}

expect class URL(spec: String) {
    open fun getHost(): String
    open fun getPath(): String
}
```

In the JVM platform code:

```
actual fun hello(world: String): String =
    "Hello, $world, on the JVM platform!"

// using existing platform-specific implementation:
actual typealias URL = java.net.URL
```

See the multiplatform programming documentation for details and steps to build a multiplatform project.

Other language features

Array literals in annotations

Starting with Kotlin 1.2, array arguments for annotations can be passed with the new array literal syntax instead of the arrayOf function:

```
@CacheConfig(cacheNames = ["books", "default"])
public class BookRepositoryImpl {
    // ...
}
```

The array literal syntax is constrained to annotation arguments.

Lateinit top-level properties and local variables

The lateinit modifier can now be used on top-level properties and local variables. The latter can be used, for example, when a lambda passed as a constructor argument to one object refers to another object which has to be defined later:

```
class Node<T>(val value: T, val next: () -> Node<T>)

fun main(args: Array<String>) {
    // A cycle of three nodes:
    lateinit var third: Node<Int>

    val second = Node(2, next = { third })
    val first = Node(1, next = { second })

    third = Node(3, next = { first })

    val nodes = generateSequence(first) { it.next() }
    println("Values in the cycle: ${nodes.take(7).joinToString { it.value.toString() }}, ...")
}
```

Check whether a lateinit var is initialized

You can now check whether a lateinit var has been initialized using isInitialized on the property reference:

```
class Foo {
    lateinit var lateinitVar: String

    fun initializationLogic() {
//sampleStart
        println("isInitialized before assignment: " + this::lateinitVar.isInitialized)
        lateinitVar = "value"
        println("isInitialized after assignment: " + this::lateinitVar.isInitialized)
//sampleEnd
    }
}

fun main(args: Array<String>) {
 Foo().initializationLogic()
}
```

Inline functions with default functional parameters

Inline functions are now allowed to have default values for their inlined functional parameters:

```
//sampleStart
inline fun <E> Iterable<E>.strings(transform: (E) -> String = { it.toString() }) =
    map { transform(it) }

val defaultStrings = listOf(1, 2, 3).strings()
val customStrings = listOf(1, 2, 3).strings { "($it)" }
//sampleEnd

fun main(args: Array<String>) {
    println("defaultStrings = $defaultStrings")
    println("customStrings = $customStrings")
}
```

Information from explicit casts is used for type inference

The Kotlin compiler can now use information from type casts in type inference. If you're calling a generic method that returns a type parameter T and casting the

return value to a specific type Foo, the compiler now understands that T for this call needs to be bound to the type Foo.

This is particularly important for Android developers, since the compiler can now correctly analyze generic findViewById calls in Android API level 26:

```
val button = findViewById(R.id.button) as Button
```

Smart cast improvements

When a variable is assigned from a safe call expression and checked for null, the smart cast is now applied to the safe call receiver as well:

```
fun countFirst(s: Any): Int {
//sampleStart
    val firstChar = (s as? CharSequence)?.firstOrNull()
    if (firstChar != null)
    return s.count { it == firstChar } // s: Any is smart cast to CharSequence

    val firstItem = (s as? Iterable<*>)?.firstOrNull()
    if (firstItem != null)
    return s.count { it == firstItem } // s: Any is smart cast to Iterable<*>
//sampleEnd
    return -1
}

fun main(args: Array<String>) {
  val string = "abacaba"
  val countInString = countFirst(string)
  println("called on \"$string\": $countInString")

  val list = listOf(1, 2, 3, 1, 2)
  val countInList = countFirst(list)
  println("called on $list: $countInList")
}
```

Also, smart casts in a lambda are now allowed for local variables that are only modified before the lambda:

```
fun main(args: Array<String>) {
//sampleStart
    val flag = args.size == 0
    var x: String? = null
    if (flag) x = "Yahoo!"

    run {
        if (x != null) {
            println(x.length) // x is smart cast to String
        }
    }
//sampleEnd
}
```

Support for ::foo as a shorthand for this::foo

A bound callable reference to a member of this can now be written without explicit receiver, ::foo instead of this::foo. This also makes callable references more convenient to use in lambdas where you refer to a member of the outer receiver.

Breaking change: sound smart casts after try blocks

Earlier, Kotlin used assignments made inside a try block for smart casts after the block, which could break type- and null-safety and lead to runtime failures. This release fixes this issue, making the smart casts more strict, but breaking some code that relied on such smart casts.

To switch to the old smart casts behavior, pass the fallback flag -Xlegacy-smart-cast-after-try as the compiler argument. It will become deprecated in Kotlin 1.3.

Deprecation: data classes overriding copy

When a data class derived from a type that already had the copy function with the same signature, the copy implementation generated for the data class used the defaults from the supertype, leading to counter-intuitive behavior, or failed at runtime if there were no default parameters in the supertype.

Inheritance that leads to a copy conflict has become deprecated with a warning in Kotlin 1.2 and will be an error in Kotlin 1.3.

Deprecation: nested types in enum entries

Inside enum entries, defining a nested type that is not an inner class has been deprecated due to issues in the initialization logic. This causes a warning in Kotlin 1.2 and will become an error in Kotlin 1.3.

Deprecation: single named argument for vararg

For consistency with array literals in annotations, passing a single item for a vararg parameter in the named form (foo(items = i)) has been deprecated. Please use the spread operator with the corresponding array factory functions:

```
foo(items = *arrayOf(1))
```

There is an optimization that removes redundant arrays creation in such cases, which prevents performance degradation. The single-argument form produces warnings in Kotlin 1.2 and is to be dropped in Kotlin 1.3.

Deprecation: inner classes of generic classes extending Throwable

Inner classes of generic types that inherit from Throwable could violate type-safety in a throw-catch scenario and thus have been deprecated, with a warning in Kotlin 1.2 and an error in Kotlin 1.3.

Deprecation: mutating backing field of a read-only property

Mutating the backing field of a read-only property by assigning field = ... in the custom getter has been deprecated, with a warning in Kotlin 1.2 and an error in Kotlin 1.3.

Standard library

Kotlin standard library artifacts and split packages

The Kotlin standard library is now fully compatible with the Java 9 module system, which forbids split packages (multiple jar files declaring classes in the same package). In order to support that, new artifacts kotlin-stdlib-jdk7 and kotlin-stdlib-jdk8 are introduced, which replace the old kotlin-stdlib-jre7 and kotlin-stdlib-jre8.

The declarations in the new artifacts are visible under the same package names from the Kotlin point of view, but have different package names for Java. Therefore, switching to the new artifacts will not require any changes to your source code.

Another change made to ensure compatibility with the new module system is removing the deprecated declarations in the kotlin.reflect package from the kotlin-reflect library. If you were using them, you need to switch to using the declarations in the kotlin.reflect.full package, which is supported since Kotlin 1.1.

windowed, chunked, zipWithNext

New extensions for Iterable<T>, Sequence<T>, and CharSequence cover such use cases as buffering or batch processing (chunked), sliding window and computing sliding average (windowed) , and processing pairs of subsequent items (zipWithNext):

```
fun main(args: Array<String>) {
//sampleStart
    val items = (1..9).map { it * it }

    val chunkedIntoLists = items.chunked(4)
    val points3d = items.chunked(3) { (x, y, z) -> Triple(x, y, z) }
    val windowed = items.windowed(4)
    val slidingAverage = items.windowed(4) { it.average() }
    val pairwiseDifferences = items.zipWithNext { a, b -> b - a }
//sampleEnd

    println("items: $items\n")

    println("chunked into lists: $chunkedIntoLists")
    println("3D points: $points3d")
    println("windowed by 4: $windowed")
    println("sliding average by 4: $slidingAverage")
    println("pairwise differences: $pairwiseDifferences")
}
```

fill, replaceAll, shuffle/shuffled

A set of extension functions was added for manipulating lists: fill, replaceAll and shuffle for MutableList, and shuffled for read-only List:

```
fun main(args: Array<String>) {
//sampleStart
    val items = (1..5).toMutableList()

    items.shuffle()
    println("Shuffled items: $items")

    items.replaceAll { it * 2 }
    println("Items doubled: $items")

    items.fill(5)
    println("Items filled with 5: $items")
//sampleEnd
}
```

Math operations in kotlin-stdlib

Satisfying the longstanding request, Kotlin 1.2 adds the kotlin.math API for math operations that is common for JVM and JS and contains the following:

- Constants: PI and E
- Trigonometric: cos, sin, tan and inverse of them: acos, asin, atan, atan2
- Hyperbolic: cosh, sinh, tanh and their inverse: acosh, asinh, atanh
- Exponentation: pow (an extension function), sqrt, hypot, exp, expm1
- Logarithms: log, log2, log10, ln, ln1p
- Rounding:
  - ceil, floor, truncate, round (half to even) functions
  - roundToInt, roundToLong (half to integer) extension functions
- Sign and absolute value:
  - abs and sign functions
  - absoluteValue and sign extension properties
  - withSign extension function
- max and min of two values
- Binary representation:
  - ulp extension property
  - nextUp, nextDown, nextTowards extension functions
  - toBits, toRawBits, Double.fromBits (these are in the kotlin package)

The same set of functions (but without constants) is also available for Float arguments.

Operators and conversions for BigInteger and BigDecimal

Kotlin 1.2 introduces a set of functions for operating with BigInteger and BigDecimal and creating them from other numeric types. These are:

- toBigInteger for Int and Long
- toBigDecimal for Int, Long, Float, Double, and BigInteger
- Arithmetic and bitwise operator functions:
  - Binary operators +, -, *, /, % and infix functions and, or, xor, shl, shr
  - Unary operators -, ++, --, and a function inv

Floating point to bits conversions

New functions were added for converting Double and Float to and from their bit representations:

- toBits and toRawBits returning Long for Double and Int for Float
- Double.fromBits and Float.fromBits for creating floating point numbers from the bit representation

Regex is now serializable

The kotlin.text.Regex class has become Serializable and can now be used in serializable hierarchies.

Closeable.use calls Throwable.addSuppressed if available

The Closeable.use function calls Throwable.addSuppressed when an exception is thrown during closing the resource after some other exception.

To enable this behavior you need to have kotlin-stdlib-jdk7 in your dependencies.

JVM backend

Constructor calls normalization

Ever since version 1.0, Kotlin supported expressions with complex control flow, such as try-catch expressions and inline function calls. Such code is valid according to the Java Virtual Machine specification. Unfortunately, some bytecode processing tools do not handle such code quite well when such expressions are present in the arguments of constructor calls.

To mitigate this problem for the users of such bytecode processing tools, we've added a command-line compiler option (-Xnormalize-constructor-calls=MODE) that tells the compiler to generate more Java-like bytecode for such constructs. Here MODE is one of:

- disable (default) – generate bytecode in the same way as in Kotlin 1.0 and 1.1.
- enable – generate Java-like bytecode for constructor calls. This can change the order in which the classes are loaded and initialized.
- preserve-class-initialization – generate Java-like bytecode for constructor calls, ensuring that the class initialization order is preserved. This can affect overall performance of your application; use it only if you have some complex state shared between multiple classes and updated on class initialization.

The "manual" workaround is to store the values of sub-expressions with control flow in variables, instead of evaluating them directly inside the call arguments. It's similar to -Xnormalize-constructor-calls=enable.

Java-default method calls

Before Kotlin 1.2, interface members overriding Java-default methods while targeting JVM 1.6 produced a warning on super calls: Super calls to Java default methods are deprecated in JVM target 1.6. Recompile with '-jvm-target 1.8'. In Kotlin 1.2, there's an error instead, thus requiring any such code to be compiled with JVM target 1.8.

Breaking change: consistent behavior of x.equals(null) for platform types

Calling x.equals(null) on a platform type that is mapped to a Java primitive (Int!, Boolean!, Short!, Long!, Float!, Double!, Char!) incorrectly returned true when x was null. Starting with Kotlin 1.2, calling x.equals(...) on a null value of a platform type throws an NPE (but x == ... does not).

To return to the pre-1.2 behavior, pass the flag -Xno-exception-on-explicit-equals-for-boxed-null to the compiler.

Breaking change: fix for platform null escaping through an inlined extension receiver

Inline extension functions that were called on a null value of a platform type did not check the receiver for null and would thus allow null to escape into the other code. Kotlin 1.2 forces this check at the call sites, throwing an exception if the receiver is null.

To switch to the old behavior, pass the fallback flag -Xno-receiver-assertions to the compiler.

JavaScript backend

TypedArrays support enabled by default

The JS typed arrays support that translates Kotlin primitive arrays, such as IntArray, DoubleArray, into JavaScript typed arrays, that was previously an opt-in feature, has been enabled by default.

Tools

Warnings as errors

The compiler now provides an option to treat all warnings as errors. Use -Werror on the command line, or the following Gradle snippet:

```
compileKotlin {
    kotlinOptions.allWarningsAsErrors = true
}
```

What's new in Kotlin 1.1

Released: 15 February 2016

Table of contents

- Coroutines
- Other language features
- Standard library
- JVM backend
- JavaScript backend

JavaScript

Starting with Kotlin 1.1, the JavaScript target is no longer considered experimental. All language features are supported, and there are many new tools for integration with the frontend development environment. See below for a more detailed list of changes.

Coroutines (experimental)

The key new feature in Kotlin 1.1 is coroutines, bringing the support of async/await, yield, and similar programming patterns. The key feature of Kotlin's design is that the implementation of coroutine execution is part of the libraries, not the language, so you aren't bound to any specific programming paradigm or concurrency library.

A coroutine is effectively a light-weight thread that can be suspended and resumed later. Coroutines are supported through suspending functions: a call to such a function can potentially suspend a coroutine, and to start a new coroutine we usually use an anonymous suspending functions (i.e. suspending lambdas).

Let's look at async/await which is implemented in an external library, kotlinx.coroutines:

```
// runs the code in the background thread pool
fun asyncOverlay() = async(CommonPool) {
    // start two async operations
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // and then apply overlay to both results
    applyOverlay(original.await(), overlay.await())
}

// launches new coroutine in UI context
launch(UI) {
    // wait for async overlay to complete
    val image = asyncOverlay().await()
    // and then show it in UI
    showImage(image)
}
```

Here, async { ... } starts a coroutine and, when we use await(), the execution of the coroutine is suspended while the operation being awaited is executed, and is resumed (possibly on a different thread) when the operation being awaited completes.

The standard library uses coroutines to support lazily generated sequences with yield and yieldAll functions. In such a sequence, the block of code that returns sequence elements is suspended after each element has been retrieved, and resumed when the next element is requested. Here's an example:

```
import kotlin.coroutines.experimental.*

fun main(args: Array<String>) {
    val seq = buildSequence {
      for (i in 1..5) {
          // yield a square of i
          yield(i * i)
      }
      // yield a range
      yieldAll(26..28)
    }

    // print the sequence
    println(seq.toList())
}
```

Run the code above to see the result. Feel free to edit it and run again!

For more information, please refer to the coroutines documentation and tutorial.

Note that coroutines are currently considered an experimental feature, meaning that the Kotlin team is not committing to supporting the backwards compatibility of this feature after the final 1.1 release.

Other language features

Type aliases

A type alias allows you to define an alternative name for an existing type. This is most useful for generic types such as collections, as well as for function types. Here is an example:

```
//sampleStart
typealias OscarWinners = Map<String, String>

fun countLaLaLand(oscarWinners: OscarWinners) =
        oscarWinners.count { it.value.contains("La La Land") }

// Note that the type names (initial and the type alias) are interchangeable:
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
        oscarWinners["Best picture"] == "La La Land"
//sampleEnd

fun oscarWinners(): OscarWinners {
    return mapOf(
            "Best song" to "City of Stars (La La Land)",
            "Best actress" to "Emma Stone (La La Land)",
            "Best picture" to "Moonlight" /* ... */)
}

fun main(args: Array<String>) {
    val oscarWinners = oscarWinners()

    val laLaLandAwards = countLaLaLand(oscarWinners)
    println("LaLaLandAwards = $laLaLandAwards (in our small example), but actually it's 6.")

    val laLaLandIsTheBestMovie = checkLaLaLandIsTheBestMovie(oscarWinners)
    println("LaLaLandIsTheBestMovie = $laLaLandIsTheBestMovie")
}
```

See the type aliases documentation and KEEP for more details.

Bound callable references

You can now use the :: operator to get a member reference pointing to a method or property of a specific object instance. Previously this could only be expressed

with a lambda. Here's an example:

```
//sampleStart
val numberRegex = "\\d+".toRegex()
val numbers = listOf("abc", "123", "456").filter(numberRegex::matches)
//sampleEnd

fun main(args: Array<String>) {
    println("Result is $numbers")
}
```

Read the documentation and KEEP for more details.

Sealed and data classes

Kotlin 1.1 removes some of the restrictions on sealed and data classes that were present in Kotlin 1.0. Now you can define subclasses of a top-level sealed class on the top level in the same file, and not just as nested classes of the sealed class. Data classes can now extend other classes. This can be used to define a hierarchy of expression classes nicely and cleanly:

```
//sampleStart
sealed class Expr

data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when (expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
}
val e = eval(Sum(Const(1.0), Const(2.0)))
//sampleEnd

fun main(args: Array<String>) {
    println("e is $e") // 3.0
}
```

Read the sealed classes documentation or KEEPs for sealed class and data class for more detail.

Destructuring in lambdas

You can now use the destructuring declaration syntax to unpack the arguments passed to a lambda. Here's an example:

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf(1 to "one", 2 to "two")
    // before
    println(map.mapValues { entry ->
      val (key, value) = entry
      "$key -> $value!"
    })
    // now
    println(map.mapValues { (key, value) -> "$key -> $value!" })
//sampleEnd
}
```

Read the destructuring declarations documentation and KEEP for more details.

Underscores for unused parameters

For a lambda with multiple parameters, you can use the _ character to replace the names of the parameters you don't use:

```
fun main(args: Array<String>) {
    val map = mapOf(1 to "one", 2 to "two")

//sampleStart
    map.forEach { _, value -> println("$value!") }
//sampleEnd
}
```

This also works in destructuring declarations:

```
data class Result(val value: Any, val status: String)

fun getResult() = Result(42, "ok").also { println("getResult() returns $it") }

fun main(args: Array<String>) {
//sampleStart
    val (_, status) = getResult()
//sampleEnd
    println("status is '$status'")
}
```

Read the KEEP for more details.

Underscores in numeric literals

Just as in Java 8, Kotlin now allows to use underscores in numeric literals to separate groups of digits:

```
//sampleStart
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
//sampleEnd

fun main(args: Array<String>) {
    println(oneMillion)
    println(hexBytes.toString(16))
    println(bytes.toString(2))
}
```

Read the KEEP for more details.

Shorter syntax for properties

For properties with the getter defined as an expression body, the property type can now be omitted:

```
//sampleStart
    data class Person(val name: String, val age: Int) {
    val isAdult get() = age >= 20 // Property type inferred to be 'Boolean'
}
//sampleEnd
fun main(args: Array<String>) {
    val akari = Person("Akari", 26)
    println("$akari.isAdult = ${akari.isAdult}")
}
```

Inline property accessors

You can now mark property accessors with the inline modifier if the properties don't have a backing field. Such accessors are compiled in the same way as inline functions.

```
//sampleStart
public val <T> List<T>.lastIndex: Int
    inline get() = this.size - 1
//sampleEnd

fun main(args: Array<String>) {
    val list = listOf('a', 'b')
    // the getter will be inlined
    println("Last index of $list is ${list.lastIndex}")
}
```

You can also mark the entire property as inline - then the modifier is applied to both accessors.

Read the inline functions documentation and KEEP for more details.

Local delegated properties

You can now use the delegated property syntax with local variables. One possible use is defining a lazily evaluated local variable:

```
import java.util.Random

fun needAnswer() = Random().nextBoolean()

fun main(args: Array<String>) {
//sampleStart
    val answer by lazy {
        println("Calculating the answer...")
        42
    }
    if (needAnswer()) {                     // returns the random value
        println("The answer is $answer.")   // answer is calculated at this point
    }
    else {
        println("Sometimes no answer is the answer...")
    }
//sampleEnd
}
```

Read the KEEP for more details.

Interception of delegated property binding

For delegated properties, it is now possible to intercept delegate to property binding using the provideDelegate operator. For example, if we want to check the property name before binding, we can write something like this:

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, prop: KProperty<*>): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        ... // property creation
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

The provideDelegate method will be called for each property during the creation of a MyUI instance, and it can perform the necessary validation right away.

Read the delegated properties documentation for more details.

Generic enum value access

It is now possible to enumerate the values of an enum class in a generic way.

```
//sampleStart
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}
//sampleEnd

fun main(args: Array<String>) {
    printAllValues<RGB>() // prints RED, GREEN, BLUE
}
```

Scope control for implicit receivers in DSLs

The @DslMarker annotation allows to restrict the use of receivers from outer scopes in a DSL context. Consider the canonical HTML builder example:

```
table {
    tr {
        td { + "Text" }
```

```
    }
}
```

In Kotlin 1.0, code in the lambda passed to td has access to three implicit receivers: the one passed to table, to tr and to td. This allows you to call methods that make no sense in the context - for example to call tr inside td and thus to put a <tr> tag in a <td>.

In Kotlin 1.1, you can restrict that, so that only methods defined on the implicit receiver of td will be available inside the lambda passed to td. You do that by defining your annotation marked with the @DslMarker meta-annotation and applying it to the base class of the tag classes.

Read the type safe builders documentation and KEEP for more details.

rem operator

The mod operator is now deprecated, and rem is used instead. See this issue for motivation.

Standard library

String to number conversions

There is a bunch of new extensions on the String class to convert it to a number without throwing an exception on invalid number: String.toIntOrNull(): Int?, String.toDoubleOrNull(): Double? etc.

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

Also integer conversion functions, like Int.toString(), String.toInt(), String.toIntOrNull(), each got an overload with radix parameter, which allows to specify the base of conversion (2 to 36).

onEach()

onEach is a small, but useful extension function for collections and sequences, which allows to perform some action, possibly with side-effects, on each element of the collection/sequence in a chain of operations. On iterables it behaves like forEach but also returns the iterable instance further. And on sequences it returns a wrapping sequence, which applies the given action lazily as the elements are being iterated.

```
inputDir.walk()
        .filter { it.isFile && it.name.endsWith(".txt") }
        .onEach { println("Moving $it to $outputDir") }
        .forEach { moveFile(it, File(outputDir, it.toRelativeString(inputDir))) }
```

also(), takeIf(), and takeUnless()

These are three general-purpose extension functions applicable to any receiver.

also is like apply: it takes the receiver, does some action on it, and returns that receiver. The difference is that in the block inside apply the receiver is available as this, while in the block inside also it's available as it (and you can give it another name if you want). This comes handy when you do not want to shadow this from the outer scope:

```
class Block {
    lateinit var content: String
}

//sampleStart
fun Block.copy() = Block().also {
    it.content = this.content
}
//sampleEnd

// using 'apply' instead
fun Block.copy1() = Block().apply {
    this.content = this@copy1.content
}

fun main(args: Array<String>) {
    val block = Block().apply { content = "content" }
    val copy = block.copy()
    println("Testing the content was copied:")
```

```
    println(block.content == copy.content)
}
```

takeIf is like filter for a single value. It checks whether the receiver meets the predicate, and returns the receiver, if it does or null if it doesn't. Combined with an elvis operator (?:) and early returns it allows writing constructs like:

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?: return false
// do something with existing outDirFile
```

```
fun main(args: Array<String>) {
    val input = "Kotlin"
    val keyword = "in"

//sampleStart
    val index = input.indexOf(keyword).takeIf { it >= 0 } ?: error("keyword not found")
    // do something with index of keyword in input string, given that it's found
//sampleEnd

    println("'$keyword' was found in '$input'")
    println(input)
    println(" ".repeat(index) + "^")
}
```

takeUnless is the same as takeIf, but it takes the inverted predicate. It returns the receiver when it doesn't meet the predicate and null otherwise. So one of the examples above could be rewritten with takeUnless as following:

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?: error("keyword not found")
```

It is also convenient to use when you have a callable reference instead of the lambda:

```
private fun testTakeUnless(string: String) {
//sampleStart
    val result = string.takeUnless(String::isEmpty)
//sampleEnd

    println("string = \"$string\"; result = \"$result\"")
}

fun main(args: Array<String>) {
    testTakeUnless("")
    testTakeUnless("abc")
}
```

groupingBy()

This API can be used to group a collection by key and fold each group simultaneously. For example, it can be used to count the number of words starting with each letter:

```
fun main(args: Array<String>) {
    val words = "one two three four five six seven eight nine ten".split(' ')
//sampleStart
    val frequencies = words.groupingBy { it.first() }.eachCount()
//sampleEnd
    println("Counting first letters: $frequencies.")

    // The alternative way that uses 'groupBy' and 'mapValues' creates an intermediate map,
    // while 'groupingBy' way counts on the fly.
    val groupBy = words.groupBy { it.first() }.mapValues { (_, list) -> list.size }
    println("Comparing the result with using 'groupBy': ${groupBy == frequencies}.")
}
```

Map.toMap() and Map.toMutableMap()

These functions can be used for easy copying of maps:

```
class ImmutablePropertyBag(map: Map<String, Any>) {
    private val mapCopy = map.toMap()
}
```

Map.minus(key)

The operator plus provides a way to add key-value pair(s) to a read-only map producing a new map, however there was not a simple way to do the opposite: to remove a key from the map you have to resort to less straightforward ways to like Map.filter() or Map.filterKeys(). Now the operator minus fills this gap. There are 4 overloads available: for removing a single key, a collection of keys, a sequence of keys and an array of keys.

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf("key" to 42)
    val emptyMap = map - "key"
//sampleEnd

    println("map: $map")
    println("emptyMap: $emptyMap")
}
```

minOf() and maxOf()

These functions can be used to find the lowest and greatest of two or three given values, where values are primitive numbers or Comparable objects. There is also an overload of each function that take an additional Comparator instance if you want to compare objects that are not comparable themselves.

```
fun main(args: Array<String>) {
//sampleStart
    val list1 = listOf("a", "b")
    val list2 = listOf("x", "y", "z")
    val minSize = minOf(list1.size, list2.size)
    val longestList = maxOf(list1, list2, compareBy { it.size })
//sampleEnd

    println("minSize = $minSize")
    println("longestList = $longestList")
}
```

Array-like List instantiation functions

Similar to the Array constructor, there are now functions that create List and MutableList instances and initialize each element by calling a lambda:

```
fun main(args: Array<String>) {
//sampleStart
    val squares = List(10) { index -> index * index }
    val mutable = MutableList(10) { 0 }
//sampleEnd

    println("squares: $squares")
    println("mutable: $mutable")
}
```

Map.getValue()

This extension on Map returns an existing value corresponding to the given key or throws an exception, mentioning which key was not found. If the map was produced with withDefault, this function will return the default value instead of throwing an exception.

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf("key" to 42)
    // returns non-nullable Int value 42
    val value: Int = map.getValue("key")

    val mapWithDefault = map.withDefault { k -> k.length }
    // returns 4
    val value2 = mapWithDefault.getValue("key2")

    // map.getValue("anotherKey") // <- this will throw NoSuchElementException
//sampleEnd

    println("value is $value")
    println("value2 is $value2")
}
```

Abstract collections

These abstract classes can be used as base classes when implementing Kotlin collection classes. For implementing read-only collections there are AbstractCollection, AbstractList, AbstractSet and AbstractMap, and for mutable collections there are AbstractMutableCollection, AbstractMutableList, AbstractMutableSet and AbstractMutableMap. On JVM, these abstract mutable collections inherit most of their functionality from JDK's abstract collections.

Array manipulation functions

The standard library now provides a set of functions for element-by-element operations on arrays: comparison (contentEquals and contentDeepEquals), hash code calculation (contentHashCode and contentDeepHashCode), and conversion to a string (contentToString and contentDeepToString). They're supported both for the JVM (where they act as aliases for the corresponding functions in java.util.Arrays) and for JS (where the implementation is provided in the Kotlin standard library).

```
fun main(args: Array<String>) {
//sampleStart
    val array = arrayOf("a", "b", "c")
    println(array.toString())  // JVM implementation: type-and-hash gibberish
    println(array.contentToString())  // nicely formatted as list
//sampleEnd
}
```

JVM Backend

Java 8 bytecode support

Kotlin has now the option of generating Java 8 bytecode (-jvm-target 1.8 command line option or the corresponding options in Ant/Maven/Gradle). For now this doesn't change the semantics of the bytecode (in particular, default methods in interfaces and lambdas are generated exactly as in Kotlin 1.0), but we plan to make further use of this later.

Java 8 standard library support

There are now separate versions of the standard library supporting the new JDK APIs added in Java 7 and 8. If you need access to the new APIs, use kotlin-stdlib-jre7 and kotlin-stdlib-jre8 maven artifacts instead of the standard kotlin-stdlib. These artifacts are tiny extensions on top of kotlin-stdlib and they bring it to your project as a transitive dependency.

Parameter names in the bytecode

Kotlin now supports storing parameter names in the bytecode. This can be enabled using the -java-parameters command line option.

Constant inlining

The compiler now inlines values of const val properties into the locations where they are used.

Mutable closure variables

The box classes used for capturing mutable closure variables in lambdas no longer have volatile fields. This change improves performance, but can lead to new race conditions in some rare usage scenarios. If you're affected by this, you need to provide your own synchronization for accessing the variables.

javax.script support

Kotlin now integrates with the javax.script API (JSR-223). The API allows to evaluate snippets of code at runtime:

```
val engine = ScriptEngineManager().getEngineByExtension("kts")!!
engine.eval("val x = 3")
println(engine.eval("x + 2"))  // Prints out 5
```

See here for a larger example project using the API.

kotlin.reflect.full

To prepare for Java 9 support, the extension functions and properties in the kotlin-reflect.jar library have been moved to the package kotlin.reflect.full. The names in

the old package (kotlin.reflect) are deprecated and will be removed in Kotlin 1.2. Note that the core reflection interfaces (such as KClass) are part of the Kotlin standard library, not kotlin-reflect, and are not affected by the move.

JavaScript backend

Unified standard library

A much larger part of the Kotlin standard library can now be used from code compiled to JavaScript. In particular, key classes such as collections (ArrayList, HashMap etc.), exceptions (IllegalArgumentException etc.) and a few others (StringBuilder, Comparator) are now defined under the kotlin package. On the JVM, the names are type aliases for the corresponding JDK classes, and on the JS, the classes are implemented in the Kotlin standard library.

Better code generation

JavaScript backend now generates more statically checkable code, which is friendlier to JS code processing tools, like minifiers, optimisers, linters, etc.

The external modifier

If you need to access a class implemented in JavaScript from Kotlin in a typesafe way, you can write a Kotlin declaration using the external modifier. (In Kotlin 1.0, the @native annotation was used instead.) Unlike the JVM target, the JS one permits to use external modifier with classes and properties. For example, here's how you can declare the DOM Node class:

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}
```

Improved import handling

You can now describe declarations which should be imported from JavaScript modules more precisely. If you add the @JsModule("<module-name>") annotation on an external declaration it will be properly imported to a module system (either CommonJS or AMD) during the compilation. For example, with CommonJS the declaration will be imported via require(...) function. Additionally, if you want to import a declaration either as a module or as a global JavaScript object, you can use the @JsNonModule annotation.

For example, here's how you can import JQuery into a Kotlin module:

```
external interface JQuery {
    fun toggle(duration: Int = definedExternally): JQuery
    fun click(handler: (Event) -> Unit): JQuery
}

@JsModule("jquery")
@JsNonModule
@JsName("$")
external fun jquery(selector: String): JQuery
```

In this case, JQuery will be imported as a module named jquery. Alternatively, it can be used as a $-object, depending on what module system Kotlin compiler is configured to use.

You can use these declarations in your application like this:

```
fun main(args: Array<String>) {
    jquery(".toggle-button").click {
        jquery(".toggle-panel").toggle(300)
    }
}
```

Kotlin releases

We ship different types of releases:

- Feature releases (1.x) that bring major changes in the language.
- Incremental releases (1.x.y) that are shipped between feature releases and include updates in the tooling, performance improvements, and bug fixes.
- Bug fix releases (1.x.yz) that include bug fixes for incremental releases.

For example, for the feature release 1.3 we had several incremental releases including 1.3.10, 1.3.20, and 1.3.70. For 1.3.70, we had 2 bug fix releases – 1.3.71 and 1.3.72.

For each incremental and feature release, we also ship several preview (EAP) versions for you to try new features before they are released. See Early Access Preview for details.

Learn more about types of Kotlin releases and their compatibility.

Update to a new release

IntelliJ IDEA and Android Studio suggest updating to a new release once it is out. When you accept the suggestion, it automatically updates the Kotlin plugin to the new version. You can check the Kotlin version in Tools | Kotlin | Configure Kotlin Plugin Updates.

If you have projects created with earlier Kotlin versions, change the Kotlin version in your projects and update kotlinx libraries if necessary.

If you are migrating to the new feature release, Kotlin plugin's migration tools will help you with the migration.

IDE support

The IDE support for the latest version of the language is available for the following versions of IntelliJ IDEA and Android Studio:

- IntelliJ IDEA:
  - Latest stable
  - Previous stable
  - Early access versions
- Android Studio:
  - Latest released version
  - Early access versions

> Learn more about the latest Kotlin-related updates in IntelliJ IDEA in the Kotlin section of the What's new in IntelliJ IDEA page.

Release details

The following table lists details of the latest Kotlin releases.

You can also use preview versions of Kotlin.

Build info	Build highlights

Build info	Build highlights
1.9.0 Released: July 6, 2023 Release on GitHub	A feature release with Kotlin K2 compiler updates, new enum class values function, new operator for open-ended ranges, preview of Gradle configuration cache in Kotlin Multiplatform, changes to Android target support in Kotlin Multiplatform, preview of custom memory allocator in Kotlin/Native. Learn more in: • What's new in Kotlin 1.9.0 • What's new in Kotlin YouTube video
1.8.22 Released: June 8, 2023 Release on GitHub	A bug fix release for Kotlin 1.8.20. Learn more about Kotlin 1.8.20 in What's new in Kotlin 1.8.20.
1.8.21 Released: April 25, 2023 Release on GitHub	A bug fix release for Kotlin 1.8.20. Learn more about Kotlin 1.8.20 in What's new in Kotlin 1.8.20. For Android Studio Flamingo and Giraffe, the Kotlin plugin 1.8.21 will be delivered with upcoming Android Studios updates.
1.8.20 Released: April 3, 2023 Release on GitHub	A feature release with Kotlin K2 compiler updates, AutoCloseable interface and Base64 encoding in stdlib, new JVM incremental compilation enabled by default, new Kotlin/Wasm compiler backend. Learn more in: • What's new in Kotlin 1.8.20 • What's new in Kotlin YouTube video
1.8.10 Released: February 2, 2023 Release on GitHub	A bug fix release for Kotlin 1.8.0. Learn more about Kotlin 1.8.0. For Android Studio Electric Eel and Flamingo, the Kotlin plugin 1.8.10 will be delivered with upcoming Android Studios updates.
1.8.0 Released: December 28, 2022 Release on GitHub	A feature release with improved kotlin-reflect performance, new recursively copy or delete directory content experimental functions for JVM, improved Objective-C/Swift interoperability. Learn more in: • What's new in Kotlin 1.8.0 • Compatibility guide for Kotlin 1.8.0

Build info	Build highlights
1.7.21 Released: November 9, 2022 Release on GitHub	A bug fix release for Kotlin 1.7.20. Learn more about Kotlin 1.7.20 in What's new in Kotlin 1.7.20. For Android Studio Dolphin, Electric Eel, and Flamingo, the Kotlin plugin 1.7.21 will be delivered with upcoming Android Studios updates.
1.7.20 Released: September 29, 2022 Release on GitHub	An incremental release with new language features, the support for several compiler plugins in the Kotlin K2 compiler, the new Kotlin/Native memory manager enabled by default, and the support for Gradle 7.1. Learn more in: • What's new in Kotlin 1.7.20 • What's new in Kotlin YouTube video • Compatibility guide for Kotlin 1.7.20 Learn more about Kotlin 1.7.20.
1.7.10 Released: July 7, 2022 Release on GitHub	A bug fix release for Kotlin 1.7.0. Learn more about Kotlin 1.7.0. For Android Studio Dolphin (213) and Android Studio Electric Eel (221), the Kotlin plugin 1.7.10 will be delivered with upcoming Android Studios updates.
1.7.0 Released: June 9, 2022 Release on GitHub	A feature release with Kotlin K2 compiler in Alpha for JVM, stabilized language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: • What's new in Kotlin 1.7.0 • What's new in Kotlin YouTube video • Compatibility guide for Kotlin 1.7.0
1.6.21 Released: April 20, 2022 Release on GitHub	A bug fix release for Kotlin 1.6.20. Learn more about Kotlin 1.6.20.

Build info	Build highlights
1.6.20 Released: April 4, 2022 Release on GitHub	An incremental release with various improvements such as: • Prototype of context receivers • Callable references to functional interface constructors • Kotlin/Native: performance improvements for the new memory manager • Multiplatform: hierarchical project structure by default • Kotlin/JS: IR compiler improvements • Gradle: compiler execution strategies Learn more about Kotlin 1.6.20.
1.6.10 Released: December 14, 2021 Release on GitHub	A bug fix release for Kotlin 1.6.0. Learn more about Kotlin 1.6.0.
1.6.0 Released: November 16, 2021 Release on GitHub	A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: • Release blog post • What's new in Kotlin 1.6.0 • Compatibility guide
1.5.32 Released: November 29, 2021 Release on GitHub	A bug fix release for Kotlin 1.5.31. Learn more about Kotlin 1.5.30.
1.5.31 Released: September 20, 2021 Release on GitHub	A bug fix release for Kotlin 1.5.30. Learn more about Kotlin 1.5.30.

Build info	Build highlights
1.5.30 Released: August 23, 2021 Release on GitHub	An incremental release with various improvements such as: • Instantiation of annotation classes on JVM • Improved opt-in requirement mechanism and type inference • Kotlin/JS IR backend in Beta • Support for Apple Silicon targets • Improved CocoaPods support • Gradle: Java toolchain support and improved daemon configuration Learn more in: • Release blog post • What's new in Kotlin 1.5.30
1.5.21 Released: July 13, 2021 Release on GitHub	A bug fix release for Kotlin 1.5.20. Learn more about Kotlin 1.5.20.
1.5.20 Released: June 24, 2021 Release on GitHub	An incremental release with various improvements such as: • String concatenation via invokedynamic on JVM by default • Improved support for Lombok and support for JSpecify • Kotlin/Native: KDoc export to Objective-C headers and faster Array.copyInto() inside one array • Gradle: caching of annotation processors' classloaders and support for the --parallel Gradle property • Aligned behavior of stdlib functions across platforms Learn more in: • Release blog post • What's new in Kotlin 1.5.20
1.5.10 Released: May 24, 2021 Release on GitHub	A bug fix release for Kotlin 1.5.0. Learn more about Kotlin 1.5.0.

Build info	Build highlights
1.5.0 Released: May 5, 2021 Release on GitHub	A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: • Release blog post • What's new in Kotlin 1.5.0 • Compatibility guide
1.4.32 Released: March 22, 2021 Release on GitHub	A bug fix release for Kotlin 1.4.30. Learn more about Kotlin 1.4.30.
1.4.31 Released: February 25, 2021 Release on GitHub	A bug fix release for Kotlin 1.4.30 Learn more about Kotlin 1.4.30.
1.4.30 Released: February 3, 2021 Release on GitHub	An incremental release with various improvements such as: • New JVM backend, now in Beta • Preview of new language features • Improved Kotlin/Native performance • Standard library API improvements Learn more in: • Release blog post • What's new in Kotlin 1.4.30
1.4.21 Released: December 7, 2020 Release on GitHub	A bug fix release for Kotlin 1.4.20 Learn more about Kotlin 1.4.20.

Build info	Build highlights
1.4.20 Released: November 23, 2020 Release on GitHub	An incremental release with various improvements such as: • Supporting new JVM features, like string concatenation via invokedynamic • Improved performance and exception handling for Kotlin Multiplatform Mobile projects • Extensions for JDK Path: Path("dir") / "file.txt" Learn more in: • Release blog post • What's new in Kotlin 1.4.20
1.4.10 Released: September 7, 2020 Release on GitHub	A bug fix release for Kotlin 1.4.0. Learn more about Kotlin 1.4.0.
1.4.0 Released: August 17, 2020 Release on GitHub	A feature release with many features and improvements that mostly focus on quality and performance. Learn more in: • Release blog post • What's new in Kotlin 1.4.0 • Compatibility guide • Migrating to Kotlin 1.4.0
1.3.72 Released: April 15, 2020 Release on GitHub	A bug fix release for Kotlin 1.3.70. Learn more about Kotlin 1.3.70.

Kotlin roadmap

Last modified on	December 2022
Next update	June 2023

Welcome to the Kotlin roadmap! Get a sneak peek into the priorities of the Kotlin Team.

Key priorities

The goal of this roadmap is to give you a big picture. Here's a list of our key projects – the most important things we focus on delivering:

- K2 compiler: a rewrite of the Kotlin compiler optimized for speed, parallelism, and unification. It will also let us introduce many anticipated language features.
- K2-based IntelliJ plugin: much faster code completion, highlighting, and search, together with a more stable code analysis.
- Kotlin Multiplatform Mobile: promote the technology to Stable by improving the toolchain stability and documentation, and ensuring compatibility guarantees.
- Experience of library authors: a set of documentation and tools helping to set up, develop, and publish Kotlin libraries.

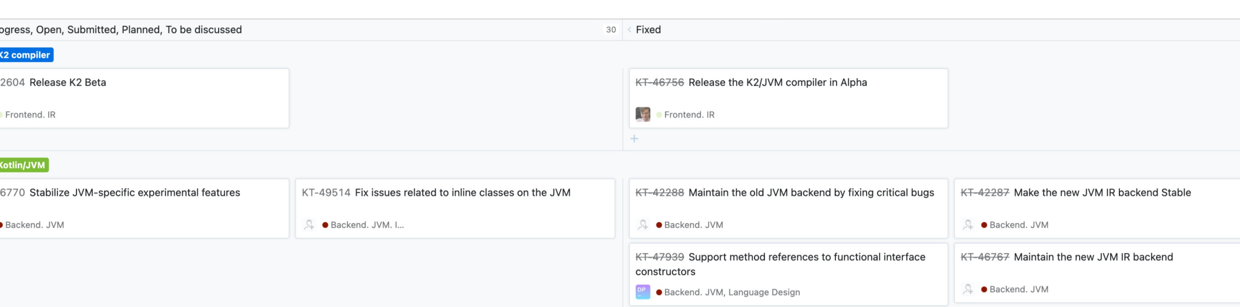
Kotlin roadmap by subsystem

To view the biggest projects we're working on, visit the YouTrack board or the Roadmap details table.

If you have any questions or feedback about the roadmap or the items on it, feel free to post them to YouTrack tickets or in the #kotlin-roadmap channel of Kotlin Slack (request an invite).

YouTrack board

Visit the roadmap board in our issue tracker YouTrack



Roadmap board in YouTrack

Roadmap details

Subsystem	In focus now
Language	List of all upcoming language features

- Introduce special syntax for until operator
- Provide modern and performant replacement for Enum.values()
- Support non-local break and continue
- Design and implement solution for toString on objects

Subsystem	In focus now
Compiler	• Release K2 Beta • Fix issues related to inline classes on the JVM • Stabilize JVM-specific experimental features • Implement an experimental version of Kotlin/Wasm compiler backend
Multiplatform	• Promote Kotlin Multiplatform Mobile to Stable • Improve the new Kotlin/Native memory manager robustness and performance and deprecate the old one • Stabilize klib: keep binary compatibility easier for library authors • Improve exporting Kotlin code to Objective-C • Improve Kotlin/Native compilation time
Tooling	• First public release of K2-based IntelliJ plugin • Improve performance and code analysis stability of the current IDE plugin • Expose stable compiler arguments in Gradle DSL • Release the Experimental version of the Kotlin Notebooks IJ IDEA plugin • Improve Kotlin scripting and experience with .gradle.kts • Provide better experience with Kotlin Daemon • Improve the performance of Gradle incremental compilation
Library ecosystem	• Improve KDoc experience • Provide a Kotlin API guide for libraries authors • Release kotlinx-metadata-jvm as Stable • Stabilize kotlinx-kover • Release kotlinx-coroutines 1.7 • Stabilize and document atomicfu • Improve kotlinx-datetime library • Continue to develop and stabilize the standard library • Release Dokka as Stable The Ktor framework roadmap

* This roadmap is not an exhaustive list of all things the team is working on, only the biggest projects.
* There's no commitment to delivering specific features or fixes in specific versions.
* We will adjust our priorities as we go and update the roadmap approximately every six months.

What's changed since May 2022

Completed items

We've completed the following items from the previous roadmap:

* Compiler core: Maintain the current compiler
* Kotlin/JVM: Support kapt in JVM IR
* Kotlin/JVM: Maintain the new JVM IR backend
* Kotlin/JVM: Improve the new JVM IR backend compilation time
* Kotlin/Native: Provide binary compatibility between incremental releases
* Kotlin/Native: Promote new memory manager to Beta and enable it by default
* Kotlin/JS: Make the new JS IR backend Stable
* Kotlin/JS: Maintain the old JS backend by fixing critical bugs
* Multiplatform: Promote Kotlin Multiplatform Mobile to Beta
* Libraries: Release kotlinx-serialization 1.4
* IDE: Stabilize code analysis
* IDE: Make update of compiler/platform versions faster
* IDE: Improve Multiplatform project support
* IDE: Stabilize Eclipse plugin
* IDE: Prototype the IDE plugin with the new compiler frontend
* IDE: Improve IDE performance
* IDE: Improve debugging experience
* Website: Make the Kotlin website mobile friendly
* Website: Make the UI and navigation consistent

New items

We've added the following items to the roadmap:

* Language: List of all upcoming language features
* Multiplatform: Promote Kotlin Multiplatform Mobile to Stable
* Multiplatform: Improve the new Kotlin/Native memory manager robustness and performance and deprecate the old one
* Tooling: First public release of K2-based IntelliJ plugin
* Tooling: Improve performance and code analysis stability of the current IDE plugin
* Tooling: Expose stable compiler arguments in Gradle DSL
* Tooling: Kotlin Notebooks IDEA plugin

- Libraries: Improve KDoc experience
- Libraries: Provide a Kotlin API guide for libraries authors

Removed items

We've removed the following items from the roadmap:

- Language: Research and prototype namespace-based solution for statics and static extensions
- Language: Multiple receivers on extension functions/properties
- Language: Support inline sealed classes
- K2 compiler: Stabilize the K2 Compiler Plugin API
- K2 compiler: Provide Alpha support for Native in the K2 platform
- K2 compiler: Provide Alpha support for JS in the K2 platform
- K2 compiler: Support Multiplatform in the K2 platform
- Multiplatform: Improve stability and robustness of the multiplatform toolchain
- Multiplatform: Improve Android support in Multiplatform projects
- Build tools: Make compilation avoidance support Stable for Gradle
- Website: Improve Kotlin Playground

Some items were removed from the roadmap but not dropped completely. In some cases, we've merged previous roadmap items with the current ones.

Items in progress

All other previously identified roadmap items are in progress. You can check their YouTrack tickets for updates.

Basic syntax

This is a collection of basic syntax elements with examples. At the end of every section, you'll find a link to a detailed description of the related topic.

You can also learn all the Kotlin essentials with the free Kotlin Core track by JetBrains Academy.

Package definition and imports

Package specification should be at the top of the source file.

```
package my.demo

import kotlin.text.*

// ...
```

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

See Packages.

Program entry point

An entry point of a Kotlin application is the main function.

```
fun main() {
    println("Hello world!")
```

```
}
```

Another form of main accepts a variable number of String arguments.

```
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

Print to the standard output

print prints its argument to the standard output.

```
fun main() {
//sampleStart
    print("Hello ")
    print("world!")
//sampleEnd
}
```

println prints its arguments and adds a line break, so that the next thing you print appears on the next line.

```
fun main() {
//sampleStart
    println("Hello world!")
    println(42)
//sampleEnd
}
```

Functions

A function with two Int parameters and Int return type.

```
//sampleStart
fun sum(a: Int, b: Int): Int {
    return a + b
}
//sampleEnd

fun main() {
    print("sum of 3 and 5 is ")
    println(sum(3, 5))
}
```

A function body can be an expression. Its return type is inferred.

```
//sampleStart
fun sum(a: Int, b: Int) = a + b
//sampleEnd

fun main() {
    println("sum of 19 and 23 is ${sum(19, 23)}")
}
```

A function that returns no meaningful value.

```
//sampleStart
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

Unit return type can be omitted.

```
//sampleStart
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

See Functions.

Variables

Read-only local variables are defined using the keyword val. They can be assigned a value only once.

```
fun main() {
//sampleStart
    val a: Int = 1  // immediate assignment
    val b = 2   // `Int` type is inferred
    val c: Int  // Type required when no initializer is provided
    c = 3       // deferred assignment
//sampleEnd
    println("a = $a, b = $b, c = $c")
}
```

Variables that can be reassigned use the var keyword.

```
fun main() {
//sampleStart
    var x = 5 // `Int` type is inferred
    x += 1
//sampleEnd
    println("x = $x")
}
```

You can declare variables at the top level.

```
//sampleStart
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
//sampleEnd

fun main() {
    println("x = $x; PI = $PI")
    incrementX()
    println("incrementX()")
    println("x = $x; PI = $PI")
}
```

See also Properties.

Creating classes and instances

To define a class, use the class keyword.

```
class Shape
```

Properties of a class can be listed in its declaration or body.

```
class Rectangle(var height: Double, var length: Double) {
```

```
    var perimeter = (height + length) * 2
}
```

The default constructor with parameters listed in the class declaration is available automatically.

```
class Rectangle(var height: Double, var length: Double) {
    var perimeter = (height + length) * 2
}
fun main() {
//sampleStart
    val rectangle = Rectangle(5.0, 2.0)
    println("The perimeter is ${rectangle.perimeter}")
//sampleEnd
}
```

Inheritance between classes is declared by a colon (:). Classes are final by default; to make a class inheritable, mark it as open.

```
open class Shape

class Rectangle(var height: Double, var length: Double): Shape() {
    var perimeter = (height + length) * 2
}
```

See classes and objects and instances.

Comments

Just like most modern languages, Kotlin supports single-line (or end-of-line) and multi-line (block) comments.

```
// This is an end-of-line comment

/* This is a block comment
   on multiple lines. */
```

Block comments in Kotlin can be nested.

```
/* The comment starts here
/* contains a nested comment */
and ends here. */
```

See Documenting Kotlin Code for information on the documentation comment syntax.

String templates

```
fun main() {
//sampleStart
    var a = 1
    // simple name in template:
    val s1 = "a is $a"

    a = 2
    // arbitrary expression in template:
    val s2 = "${s1.replace("is", "was")}, but now is $a"
//sampleEnd
    println(s2)
}
```

See String templates for details.

Conditional expressions

```
//sampleStart
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
```

```
        return a
    } else {
        return b
    }
}
//sampleEnd

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

In Kotlin, if can also be used as an expression.

```
//sampleStart
fun maxOf(a: Int, b: Int) = if (a > b) a else b
//sampleEnd

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

See if-expressions.

for loop

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

or

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
//sampleEnd
}
```

See for loop.

while loop

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    var index = 0
    while (index < items.size) {
        println("item at $index is ${items[index]}")
        index++
    }
//sampleEnd
}
```

See while loop.

when expression

```
//sampleStart
fun describe(obj: Any): String =
```

```
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
//sampleEnd

fun main() {
    println(describe(1))
    println(describe("Hello"))
    println(describe(1000L))
    println(describe(2))
    println(describe("other"))
}
```

See when expression.

Ranges

Check if a number is within a range using in operator.

```
fun main() {
//sampleStart
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
//sampleEnd
}
```

Check if a number is out of range.

```
fun main() {
//sampleStart
    val list = listOf("a", "b", "c")

    if (-1 !in 0..list.lastIndex) {
        println("-1 is out of range")
    }
    if (list.size !in list.indices) {
        println("list size is out of valid list indices range, too")
    }
//sampleEnd
}
```

Iterate over a range.

```
fun main() {
//sampleStart
    for (x in 1..5) {
        print(x)
    }
//sampleEnd
}
```

Or over a progression.

```
fun main() {
//sampleStart
    for (x in 1..10 step 2) {
        print(x)
    }
    println()
    for (x in 9 downTo 0 step 3) {
        print(x)
    }
//sampleEnd
}
```

See Ranges and progressions.

Collections

Iterate over a collection.

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
//sampleStart
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

Check if a collection contains an object using in operator.

```
fun main() {
    val items = setOf("apple", "banana", "kiwifruit")
//sampleStart
    when {
        "orange" in items -> println("juicy")
        "apple" in items -> println("apple is fine too")
    }
//sampleEnd
}
```

Using lambda expressions to filter and map collections:

```
fun main() {
//sampleStart
    val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
    fruits
      .filter { it.startsWith("a") }
      .sortedBy { it }
      .map { it.uppercase() }
      .forEach { println(it) }
//sampleEnd
}
```

See Collections overview.

Nullable values and null checks

A reference must be explicitly marked as nullable when null value is possible. Nullable type names have ? at the end.

Return null if str does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Use a function returning nullable value:

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

//sampleStart
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
```

```
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
//sampleEnd

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("a", "b")
}
```

or

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

//sampleStart
    // ...
    if (x == null) {
        println("Wrong number format in arg1: '$arg1'")
        return
    }
    if (y == null) {
        println("Wrong number format in arg2: '$arg2'")
        return
    }

    // x and y are automatically cast to non-nullable after null check
    println(x * y)
//sampleEnd
}

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("99", "b")
}
```

See Null-safety.

Type checks and automatic casts

The is operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Error: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

or

```kotlin
//sampleStart
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` is automatically cast to `String` in this branch
    return obj.length
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Error: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

or even

```kotlin
//sampleStart
fun getStringLength(obj: Any): Int? {
    // `obj` is automatically cast to `String` on the right-hand side of `&&`
    if (obj is String && obj.length > 0) {
        return obj.length
    }

    return null
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Error: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength("")
    printLength(1000)
}
```

See Classes and Type casts.

Idioms

A collection of random and frequently used idioms in Kotlin. If you have a favorite idiom, contribute it by sending a pull request.

Create DTOs (POJOs/POCOs)

```kotlin
data class Customer(val name: String, val email: String)
```

provides a Customer class with the following functionality:

- getters (and setters in case of vars) for all properties
- equals()
- hashCode()
- toString()
- copy()
- component1(), component2(), ..., for all properties (see Data classes)

Default values for function parameters

```kotlin
fun foo(a: Int = 0, b: String = "") { ... }
```

Filter a list

```kotlin
val positives = list.filter { x -> x > 0 }
```

Or alternatively, even shorter:

```kotlin
val positives = list.filter { it > 0 }
```

Learn the difference between Java and Kotlin filtering.

Check the presence of an element in a collection

```kotlin
if ("john@example.com" in emailsList) { ... }

if ("jane@example.com" !in emailsList) { ... }
```

String interpolation

```kotlin
println("Name $name")
```

Learn the difference between Java and Kotlin string concatenation.

Instance checks

```kotlin
when (x) {
    is Foo -> ...
    is Bar -> ...
    else   -> ...
}
```

Read-only list

```kotlin
val list = listOf("a", "b", "c")
```

Read-only map

```kotlin
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

Access a map entry

```kotlin
println(map["key"])
map["key"] = value
```

Traverse a map or a list of pairs

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

k and v can be any convenient names, such as name and age.

Iterate over a range

```
for (i in 1..100) { ... }  // closed-ended range: includes 100
for (i in 1..<100) { ... } // open-ended range: does not include 100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
(1..10).forEach { ... }
```

Lazy property

```
val p: String by lazy { // the value is computed only on first access
    // compute the string
}
```

Extension functions

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

Create a singleton

```
object Resource {
    val name = "Name"
}
```

Instantiate an abstract class

```
abstract class MyAbstractClass {
    abstract fun doSomething()
    abstract fun sleep()
}

fun main() {
    val myObject = object : MyAbstractClass() {
        override fun doSomething() {
            // ...
        }

        override fun sleep() { // ...
        }
    }
    myObject.doSomething()
}
```

If-not-null shorthand

```
val files = File("Test").listFiles()

println(files?.size) // size is printed if files is not null
```

If-not-null-else shorthand

```
val files = File("Test").listFiles()

println(files?.size ?: "empty") // if files is null, this prints "empty"

// To calculate the fallback value in a code block, use `run`
val filesSize = files?.size ?: run {
    return someSize
}
println(filesSize)
```

Execute a statement if null

```
val values = ...
val email = values["email"] ?: throw IllegalStateException("Email is missing!")
```

Get first item of a possibly empty collection

```
val emails = ... // might be empty
val mainEmail = emails.firstOrNull() ?: ""
```

Learn the difference between Java and Kotlin first item getting.

Execute if not null

```
val value = ...

value?.let {
    ... // execute this block if not null
}
```

Map nullable value if not null

```
val value = ...

val mapped = value?.let { transformValue(it) } ?: defaultValue
// defaultValue is returned if the value or the transform result is null.
```

Return on when statement

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

try-catch expression

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
```

```
        throw IllegalStateException(e)
    }

    // Working with result
}
```

if expression

```
val y = if (x == 1) {
    "one"
} else if (x == 2) {
    "two"
} else {
    "other"
}
```

Builder-style usage of methods that return Unit

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

Single-expression functions

```
fun theAnswer() = 42
```

This is equivalent to

```
fun theAnswer(): Int {
    return 42
}
```

This can be effectively combined with other idioms, leading to shorter code. For example, with the when expression:

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

Call multiple methods on an object instance (with)

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { //draw a 100 pix square
    penDown()
    for (i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

Configure properties of an object (apply)

```
val myRectangle = Rectangle().apply {
    length = 4
    breadth = 5
    color = 0xFAFAFA
}
```

This is useful for configuring properties that aren't present in the object constructor.

Java 7's try-with-resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

Generic function that requires the generic type information

```
//  public final class Gson {
//     ...
//     public <T> T fromJson(JsonElement json, Class<T> classOfT) throws JsonSyntaxException {
//     ...

inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(json, T::class.java)
```

Nullable Boolean

```
val b: Boolean? = ...
if (b == true) {
    ...
} else {
    // `b` is false or null
}
```

Swap two variables

```
var a = 1
var b = 2
a = b.also { b = a }
```

Mark code as incomplete (TODO)

Kotlin's standard library has a TODO() function that will always throw a NotImplementedError. Its return type is Nothing so it can be used regardless of expected type. There's also an overload that accepts a reason parameter:

```
fun calcTaxes(): BigDecimal = TODO("Waiting for feedback from accounting")
```

IntelliJ IDEA's kotlin plugin understands the semantics of TODO() and automatically adds a code pointer in the TODO tool window.

What's next?

- Solve Advent of Code puzzles using the idiomatic Kotlin style.
- Learn how to perform typical tasks with strings in Java and Kotlin.

- Learn how to perform typical tasks with collections in Java and Kotlin.
- Learn how to handle nullability in Java and Kotlin.

Coding conventions

Commonly known and easy-to-follow coding conventions are vital for any programming language. Here we provide guidelines on the code style and code organization for projects that use Kotlin.

Configure style in IDE

Two most popular IDEs for Kotlin - IntelliJ IDEA and Android Studio provide powerful support for code styling. You can configure them to automatically format your code in consistence with the given code style.

Apply the style guide

1. Go to Settings/Preferences | Editor | Code Style | Kotlin.
2. Click Set from....
3. Select Kotlin style guide.

Verify that your code follows the style guide

1. Go to Settings/Preferences | Editor | Inspections | General.
2. Switch on Incorrect formatting inspection. Additional inspections that verify other issues described in the style guide (such as naming conventions) are enabled by default.

Source code organization

Directory structure

In pure Kotlin projects, the recommended directory structure follows the package structure with the common root package omitted. For example, if all the code in the project is in the org.example.kotlin package and its subpackages, files with the org.example.kotlin package should be placed directly under the source root, and files in org.example.kotlin.network.socket should be in the network/socket subdirectory of the source root.

> On JVM: In projects where Kotlin is used together with Java, Kotlin source files should reside in the same source root as the Java source files, and follow the same directory structure: each file should be stored in the directory corresponding to each package statement.

Source file names

If a Kotlin file contains a single class or interface (potentially with related top-level declarations), its name should be the same as the name of the class, with the .kt extension appended. It applies to all types of classes and interfaces. If a file contains multiple classes, or only top-level declarations, choose a name describing what the file contains, and name the file accordingly. Use an upper camel case with an uppercase first letter (also known as Pascal case), for example, ProcessDeclarations.kt.

The name of the file should describe what the code in the file does. Therefore, you should avoid using meaningless words such as Util in file names.

Multiplatform projects

In multiplatform projects, files with top-level declarations in platform-specific source sets should have a suffix associated with the name of the source set. For example:

- jvmMain/kotlin/Platform.jvm.kt
- androidMain/kotlin/Platform.android.kt

- iosMain/kotlin/Platform.ios.kt

As for the common source set, files with top-level declarations should not have a suffix. For example, commonMain/kotlin/Platform.kt.

Technical details

We recommend following this file naming scheme in multiplatform projects due to JVM limitations: it doesn't allow top-level members (functions, properties).

To work around this, the Kotlin JVM compiler creates wrapper classes (so-called "file facades") that contain top-level member declarations. File facades have an internal name derived from the file name.

In turn, JVM doesn't allow several classes with the same fully qualified name (FQN). This might lead to situations when a Kotlin project cannot be compiled to JVM:

```
root
|- commonMain/kotlin/myPackage/Platform.kt // contains 'fun count() { }'
|- jvmMain/kotlin/myPackage/Platform.kt // contains 'fun multiply() { }'
```

Here both Platform.kt files are in the same package, so the Kotlin JVM compiler produces two file facades, both of which have FQN myPackage.PlatformKt. This produces the "Duplicate JVM classes" error.

The simplest way to avoid that is renaming one of the files according to the guideline above. This naming scheme helps avoid clashes while retaining code readability.

> There are two cases when these recommendations may seem redundant, but we still advise to follow them:
>
> - Non-JVM platforms don't have issues with duplicating file facades. However, this naming scheme can help you keep file naming consistent.
> - On JVM, if source files don't have top-level declarations, the file facades aren't generated, and you won't face naming clashes.
>
>   However, this naming scheme can help you avoid situations when a simple refactoring or an addition could include a top-level function and result in the same "Duplicate JVM classes" error.

Source file organization

Placing multiple declarations (classes, top-level functions or properties) in the same Kotlin source file is encouraged as long as these declarations are closely related to each other semantically, and the file size remains reasonable (not exceeding a few hundred lines).

In particular, when defining extension functions for a class which are relevant for all clients of this class, put them in the same file with the class itself. When defining extension functions that make sense only for a specific client, put them next to the code of that client. Avoid creating files just to hold all extensions of some class.

Class layout

The contents of a class should go in the following order:

1. Property declarations and initializer blocks
2. Secondary constructors
3. Method declarations
4. Companion object

Do not sort the method declarations alphabetically or by visibility, and do not separate regular methods from extension methods. Instead, put related stuff together, so that someone reading the class from top to bottom can follow the logic of what's happening. Choose an order (either higher-level stuff first, or vice versa) and stick to it.

Put nested classes next to the code that uses those classes. If the classes are intended to be used externally and aren't referenced inside the class, put them in the end, after the companion object.

Interface implementation layout

When implementing an interface, keep the implementing members in the same order as members of the interface (if necessary, interspersed with additional private methods used for the implementation).

Overload layout

Always put overloads next to each other in a class.

Naming rules

Package and class naming rules in Kotlin are quite simple:

- Names of packages are always lowercase and do not use underscores (org.example.project). Using multi-word names is generally discouraged, but if you do need to use multiple words, you can either just concatenate them together or use the camel case (org.example.myProject).
- Names of classes and objects start with an uppercase letter and use the camel case:

```
open class DeclarationProcessor { /*...*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /*...*/ }
```

Function names

Names of functions, properties and local variables start with a lowercase letter and use the camel case and no underscores:

```
fun processDeclarations() { /*...*/ }
var declarationCount = 1
```

Exception: factory functions used to create instances of classes can have the same name as the abstract return type:

```
interface Foo { /*...*/ }

class FooImpl : Foo { /*...*/ }

fun Foo(): Foo { return FooImpl() }
```

Names for test methods

In tests (and only in tests), you can use method names with spaces enclosed in backticks. Note that such method names are currently not supported by the Android runtime. Underscores in method names are also allowed in test code.

```
class MyTestCase {
     @Test fun `ensure everything works`() { /*...*/ }

     @Test fun ensureEverythingWorks_onAndroid() { /*...*/ }
}
```

Property names

Names of constants (properties marked with const, or top-level or object val properties with no custom get function that hold deeply immutable data) should use uppercase underscore-separated (screaming snake case) names:

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

Names of top-level or object properties which hold objects with behavior or mutable data should use camel case names:

```
val mutableCollection: MutableSet<String> = HashSet()
```

Names of properties holding references to singleton objects can use the same naming style as object declarations:

```
val PersonComparator: Comparator<Person> = /*...*/
```

For enum constants, it's OK to use either uppercase underscore-separated names (screaming snake case) (enum class Color { RED, GREEN }) or upper camel case names, depending on the usage.

Names for backing properties

If a class has two properties which are conceptually the same but one is part of a public API and another is an implementation detail, use an underscore as the prefix for the name of the private property:

```
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
         get() = _elementList
}
```

Choose good names

The name of a class is usually a noun or a noun phrase explaining what the class is: List, PersonReader.

The name of a method is usually a verb or a verb phrase saying what the method does: close, readPersons. The name should also suggest if the method is mutating the object or returning a new one. For instance sort is sorting a collection in place, while sorted is returning a sorted copy of the collection.

The names should make it clear what the purpose of the entity is, so it's best to avoid using meaningless words (Manager, Wrapper) in names.

When using an acronym as part of a declaration name, capitalize it if it consists of two letters (IOStream); capitalize only the first letter if it is longer (XmlFormatter, HttpInputStream).

Formatting

Indentation

Use four spaces for indentation. Do not use tabs.

For curly braces, put the opening brace at the end of the line where the construct begins, and the closing brace on a separate line aligned horizontally with the opening construct.

```
if (elements != null) {
    for (element in elements) {
        // ...
    }
}
```

> In Kotlin, semicolons are optional, and therefore line breaks are significant. The language design assumes Java-style braces, and you may encounter surprising behavior if you try to use a different formatting style.

Horizontal whitespace

- Put spaces around binary operators (a + b). Exception: don't put spaces around the "range to" operator (0..i).
- Do not put spaces around unary operators (a++).
- Put spaces between control flow keywords (if, when, for, and while) and the corresponding opening parenthesis.
- Do not put a space before an opening parenthesis in a primary constructor declaration, method declaration or method call.

```
class A(val x: Int)

fun foo(x: Int) { ... }

fun bar() {
    foo(1)
}
```

- Never put a space after (, [, or before],)
- Never put a space around . or ?.: foo.bar().filter { it > 2 }.joinToString(), foo?.bar()
- Put a space after //: // This is a comment

- Do not put spaces around angle brackets used to specify type parameters: class Map<K, V> { ... }
- Do not put spaces around ::: Foo::class, String::length
- Do not put a space before ? used to mark a nullable type: String?

As a general rule, avoid horizontal alignment of any kind. Renaming an identifier to a name with a different length should not affect the formatting of either the declaration or any of the usages.

Colon

Put a space before : in the following cases:

- when it's used to separate a type and a supertype
- when delegating to a superclass constructor or a different constructor of the same class
- after the object keyword

Don't put a space before : when it separates a declaration and its type.

Always put a space after :.

```
abstract class Foo<out T : Any> : IFoo {
    abstract fun foo(a: Int): T
}

class FooImpl : Foo() {
    constructor(x: String) : this(x) { /*...*/ }

    val x = object : IFoo { /*...*/ }
}
```

Class headers

Classes with a few primary constructor parameters can be written in a single line:

```
class Person(id: Int, name: String)
```

Classes with longer headers should be formatted so that each primary constructor parameter is in a separate line with indentation. Also, the closing parenthesis should be on a new line. If you use inheritance, the superclass constructor call, or the list of implemented interfaces should be located on the same line as the parenthesis:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) { /*...*/ }
```

For multiple interfaces, the superclass constructor call should be located first and then each interface should be located in a different line:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker { /*...*/ }
```

For classes with a long supertype list, put a line break after the colon and align all supertype names horizontally:

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne {

    fun foo() { /*...*/ }
}
```

To clearly separate the class header and body when the class header is long, either put a blank line following the class header (as in the example above), or put the opening curly brace on a separate line:

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne
{
    fun foo() { /*...*/ }
}
```

Use regular indent (four spaces) for constructor parameters. This ensures that properties declared in the primary constructor have the same indentation as properties declared in the body of a class.

Modifiers order

If a declaration has multiple modifiers, always put them in the following order:

```
public / protected / private / internal
expect / actual
final / open / abstract / sealed / const
external
override
lateinit
tailrec
vararg
suspend
inner
enum / annotation / fun // as a modifier in `fun interface`
companion
inline / value
infix
operator
data
```

Place all annotations before modifiers:

```
@Named("Foo")
private val foo: Foo
```

Unless you're working on a library, omit redundant modifiers (for example, public).

Annotations

Place annotations on separate lines before the declaration to which they are attached, and with the same indentation:

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

Annotations without arguments may be placed on the same line:

```
@JsonExclude @JvmField
var x: String
```

A single annotation without arguments may be placed on the same line as the corresponding declaration:

```
@Test fun foo() { /*...*/ }
```

File annotations

File annotations are placed after the file comment (if any), before the package statement, and are separated from package with a blank line (to emphasize the fact that they target the file and not the package).

```
/** License, copyright and whatever */
@file:JvmName("FooBar")
```

```
package foo.bar
```

Functions

If the function signature doesn't fit on a single line, use the following syntax:

```
fun longMethodName(
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType,
): ReturnType {
    // body
}
```

Use regular indent (four spaces) for function parameters. It helps ensure consistency with constructor parameters.

Prefer using an expression body for functions with the body consisting of a single expression.

```
fun foo(): Int {     // bad
    return 1
}

fun foo() = 1        // good
```

Expression bodies

If the function has an expression body whose first line doesn't fit on the same line as the declaration, put the = sign on the first line and indent the expression body by four spaces.

```
fun f(x: String, y: String, z: String) =
    veryLongFunctionCallWithManyWords(andLongParametersToo(), x, y, z)
```

Properties

For very simple read-only properties, consider one-line formatting:

```
val isEmpty: Boolean get() = size == 0
```

For more complex properties, always put get and set keywords on separate lines:

```
val foo: String
    get() { /*...*/ }
```

For properties with an initializer, if the initializer is long, add a line break after the = sign and indent the initializer by four spaces:

```
private val defaultCharset: Charset? =
    EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)
```

Control flow statements

If the condition of an if or when statement is multiline, always use curly braces around the body of the statement. Indent each subsequent line of the condition by four spaces relative to the statement start. Put the closing parentheses of the condition together with the opening curly brace on a separate line:

```
if (!component.isSyncing &&
    !hasAnyKotlinRuntimeInScope(module)
) {
    return createKotlinNotConfiguredPanel(module)
}
```

This helps align the condition and statement bodies.

Put the else, catch, finally keywords, as well as the while keyword of a do-while loop, on the same line as the preceding curly brace:

```
if (condition) {
    // body
} else {
    // else part
}

try {
    // body
} finally {
    // cleanup
}
```

In a when statement, if a branch is more than a single line, consider separating it from adjacent case blocks with a blank line:

```
private fun parsePropertyValue(propName: String, token: Token) {
    when (token) {
        is Token.ValueToken ->
            callback.visitValue(propName, token.value)

        Token.LBRACE -> { // ...
        }
    }
}
```

Put short branches on the same line as the condition, without braces.

```
when (foo) {
    true -> bar() // good
    false -> { baz() } // bad
}
```

Method calls

In long argument lists, put a line break after the opening parenthesis. Indent arguments by four spaces. Group multiple closely related arguments on the same line.

```
drawSquare(
    x = 10, y = 10,
    width = 100, height = 100,
    fill = true
)
```

Put spaces around the = sign separating the argument name and value.

Wrap chained calls

When wrapping chained calls, put the . character or the ?. operator on the next line, with a single indent:

```
val anchor = owner
    ?.firstChild!!
    .siblings(forward = true)
    .dropWhile { it is PsiComment || it is PsiWhiteSpace }
```

The first call in the chain should usually have a line break before it, but it's OK to omit it if the code makes more sense that way.

Lambdas

In lambda expressions, spaces should be used around the curly braces, as well as around the arrow which separates the parameters from the body. If a call takes a single lambda, pass it outside parentheses whenever possible.

```
list.filter { it > 10 }
```

If assigning a label for a lambda, do not put a space between the label and the opening curly brace:

```
fun foo() {
    ints.forEach lit@{
        // ...
```

```
    }
}
```

When declaring parameter names in a multiline lambda, put the names on the first line, followed by the arrow and the newline:

```
appendCommaSeparated(properties) { prop ->
    val propertyValue = prop.get(obj)  // ...
}
```

If the parameter list is too long to fit on a line, put the arrow on a separate line:

```
foo {
   context: Context,
   environment: Env
   ->
   context.configureEnv(environment)
}
```

Trailing commas

A trailing comma is a comma symbol after the last item in a series of elements:

```
class Person(
    val firstName: String,
    val lastName: String,
    val age: Int, // trailing comma
)
```

Using trailing commas has several benefits:

- It makes version-control diffs cleaner – as all the focus is on the changed value.
- It makes it easy to add and reorder elements – there is no need to add or delete the comma if you manipulate elements.
- It simplifies code generation, for example, for object initializers. The last element can also have a comma.

Trailing commas are entirely optional – your code will still work without them. The Kotlin style guide encourages the use of trailing commas at the declaration site and leaves it at your discretion for the call site.

To enable trailing commas in the IntelliJ IDEA formatter, go to Settings/Preferences | Editor | Code Style | Kotlin, open the Other tab and select the Use trailing comma option.

Enumerations

```
enum class Direction {
    NORTH,
    SOUTH,
    WEST,
    EAST, // trailing comma
}
```

Value arguments

```
fun shift(x: Int, y: Int) { /*...*/ }
shift(
    25,
    20, // trailing comma
)
val colors = listOf(
    "red",
    "green",
    "blue", // trailing comma
)
```

Class properties and parameters

```kotlin
class Customer(
    val name: String,
    val lastName: String, // trailing comma
)
class Customer(
    val name: String,
    lastName: String, // trailing comma
)
```

Function value parameters

```kotlin
fun powerOf(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }
constructor(
    x: Comparable<Number>,
    y: Iterable<Number>, // trailing comma
) {}
fun print(
    vararg quantity: Int,
    description: String, // trailing comma
) {}
```

Parameters with optional type (including setters)

```kotlin
val sum: (Int, Int, Int) -> Int = fun(
    x,
    y,
    z, // trailing comma
): Int {
    return x + y + x
}
println(sum(8, 8, 8))
```

Indexing suffix

```kotlin
class Surface {
    operator fun get(x: Int, y: Int) = 2 * x + 4 * y - 10
}
fun getZValue(mySurface: Surface, xValue: Int, yValue: Int) =
    mySurface[
        xValue,
        yValue, // trailing comma
    ]
```

Parameters in lambdas

```kotlin
fun main() {
    val x = {
            x: Comparable<Number>,
            y: Iterable<Number>, // trailing comma
        ->
        println("1")
    }
    println(x)
}
```

when entry

```kotlin
fun isReferenceApplicable(myReference: KClass<*>) = when (myReference) {
    Comparable::class,
    Iterable::class,
    String::class, // trailing comma
        -> true
```

```
    else -> false
}
```

Collection literals (in annotations)

```
annotation class ApplicableFor(val services: Array<String>)
@ApplicableFor([
    "serializer",
    "balancer",
    "database",
    "inMemoryCache", // trailing comma
])
fun run() {}
```

Type arguments

```
fun <T1, T2> foo() {}
fun main() {
    foo<
            Comparable<Number>,
            Iterable<Number>, // trailing comma
            >()
}
```

Type parameters

```
class MyMap<
        MyKey,
        MyValue, // trailing comma
        > {}
```

Destructuring declarations

```
data class Car(val manufacturer: String, val model: String, val year: Int)
val myCar = Car("Tesla", "Y", 2019)
val (
    manufacturer,
    model,
    year, // trailing comma
) = myCar
val cars = listOf<Car>()
fun printMeanValue() {
    var meanValue: Int = 0
    for ((
        _,
        _,
        year, // trailing comma
    ) in cars) {
        meanValue += year
    }
    println(meanValue/cars.size)
}
printMeanValue()
```

Documentation comments

For longer documentation comments, place the opening /** on a separate line and begin each subsequent line with an asterisk:

```
/**
 * This is a documentation comment
 * on multiple lines.
 */
```

Short comments can be placed on a single line:

```
/** This is a short documentation comment. */
```

Generally, avoid using @param and @return tags. Instead, incorporate the description of parameters and return values directly into the documentation comment, and add links to parameters wherever they are mentioned. Use @param and @return only when a lengthy description is required which doesn't fit into the flow of the main text.

```
// Avoid doing this:

/**
 * Returns the absolute value of the given number.
 * @param number The number to return the absolute value for.
 * @return The absolute value.
 */
fun abs(number: Int): Int { /*...*/ }

// Do this instead:

/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: Int): Int { /*...*/ }
```

Avoid redundant constructs

In general, if a certain syntactic construction in Kotlin is optional and highlighted by the IDE as redundant, you should omit it in your code. Do not leave unnecessary syntactic elements in code just "for clarity".

Unit return type

If a function returns Unit, the return type should be omitted:

```
fun foo() { // ": Unit" is omitted here

}
```

Semicolons

Omit semicolons whenever possible.

String templates

Don't use curly braces when inserting a simple variable into a string template. Use curly braces only for longer expressions.

```
println("$name has ${children.size} children")
```

Idiomatic use of language features

Immutability

Prefer using immutable data to mutable. Always declare local variables and properties as val rather than var if they are not modified after initialization.

Always use immutable collection interfaces (Collection, List, Set, Map) to declare collections which are not mutated. When using factory functions to create collection instances, always use functions that return immutable collection types when possible:

```
// Bad: use of a mutable collection type for value which will not be mutated
fun validateValue(actualValue: String, allowedValues: HashSet<String>) { ... }

// Good: immutable collection type used instead
fun validateValue(actualValue: String, allowedValues: Set<String>) { ... }

// Bad: arrayListOf() returns ArrayList<T>, which is a mutable collection type
val allowedValues = arrayListOf("a", "b", "c")
```

```
// Good: listOf() returns List<T>
val allowedValues = listOf("a", "b", "c")
```

Default parameter values

Prefer declaring functions with default parameter values to declaring overloaded functions.

```
// Bad
fun foo() = foo("a")
fun foo(a: String) { /*...*/ }

// Good
fun foo(a: String = "a") { /*...*/ }
```

Type aliases

If you have a functional type or a type with type parameters which is used multiple times in a codebase, prefer defining a type alias for it:

```
typealias MouseClickHandler = (Any, MouseEvent) -> Unit
typealias PersonIndex = Map<String, Person>
```

If you use a private or internal type alias for avoiding name collision, prefer the import ... as ... mentioned in Packages and Imports.

Lambda parameters

In lambdas which are short and not nested, it's recommended to use the it convention instead of declaring the parameter explicitly. In nested lambdas with parameters, always declare parameters explicitly.

Returns in a lambda

Avoid using multiple labeled returns in a lambda. Consider restructuring the lambda so that it will have a single exit point. If that's not possible or not clear enough, consider converting the lambda into an anonymous function.

Do not use a labeled return for the last statement in a lambda.

Named arguments

Use the named argument syntax when a method takes multiple parameters of the same primitive type, or for parameters of Boolean type, unless the meaning of all parameters is absolutely clear from context.

```
drawSquare(x = 10, y = 10, width = 100, height = 100, fill = true)
```

Conditional statements

Prefer using the expression form of try, if, and when.

```
return if (x) foo() else bar()
```

```
return when(x) {
    0 -> "zero"
    else -> "nonzero"
}
```

The above is preferable to:

```
if (x)
    return foo()
else
    return bar()
```

```
when(x) {
    0 -> return "zero"
    else -> return "nonzero"
}
```

if versus when

Prefer using if for binary conditions instead of when. For example, use this syntax with if:

```
if (x == null) ... else ...
```

instead of this one with when:

```
when (x) {
    null -> // ...
    else -> // ...
}
```

Prefer using when if there are three or more options.

Nullable Boolean values in conditions

If you need to use a nullable Boolean in a conditional statement, use if (value == true) or if (value == false) checks.

Loops

Prefer using higher-order functions (filter, map etc.) to loops. Exception: forEach (prefer using a regular for loop instead, unless the receiver of forEach is nullable or forEach is used as part of a longer call chain).

When making a choice between a complex expression using multiple higher-order functions and a loop, understand the cost of the operations being performed in each case and keep performance considerations in mind.

Loops on ranges

Use the ..< operator to loop over an open-ended range:

```
for (i in 0..n - 1) { /*...*/ }  // bad
for (i in 0..<n) { /*...*/ }  // good
```

Strings

Prefer string templates to string concatenation.

Prefer multiline strings to embedding \n escape sequences into regular string literals.

To maintain indentation in multiline strings, use trimIndent when the resulting string does not require any internal indentation, or trimMargin when internal indentation is required:

```
fun main() {
//sampleStart
   println("""
    Not
    trimmed
    text
    """
   )

   println("""
    Trimmed
    text
    """.trimIndent()
   )

   println()

   val a = """Trimmed to margin text:
```

```
        |if(a > 1) {
        |    return a
        |}""".trimMargin()

    println(a)
//sampleEnd
}
```

Learn the difference between Java and Kotlin multiline strings.

Functions vs properties

In some cases, functions with no arguments might be interchangeable with read-only properties. Although the semantics are similar, there are some stylistic conventions on when to prefer one to another.

Prefer a property over a function when the underlying algorithm:

- does not throw
- is cheap to calculate (or cached on the first run)
- returns the same result over invocations if the object state hasn't changed

Extension functions

Use extension functions liberally. Every time you have a function that works primarily on an object, consider making it an extension function accepting that object as a receiver. To minimize API pollution, restrict the visibility of extension functions as much as it makes sense. As necessary, use local extension functions, member extension functions, or top-level extension functions with private visibility.

Infix functions

Declare a function as infix only when it works on two objects which play a similar role. Good examples: and, to, zip. Bad example: add.

Do not declare a method as infix if it mutates the receiver object.

Factory functions

If you declare a factory function for a class, avoid giving it the same name as the class itself. Prefer using a distinct name, making it clear why the behavior of the factory function is special. Only if there is really no special semantics, you can use the same name as the class.

```
class Point(val x: Double, val y: Double) {
    companion object {
        fun fromPolar(angle: Double, radius: Double) = Point(...)
    }
}
```

If you have an object with multiple overloaded constructors that don't call different superclass constructors and can't be reduced to a single constructor with default argument values, prefer to replace the overloaded constructors with factory functions.

Platform types

A public function/method returning an expression of a platform type must declare its Kotlin type explicitly:

```
fun apiCall(): String = MyJavaApi.getProperty("name")
```

Any property (package-level or class-level) initialized with an expression of a platform type must declare its Kotlin type explicitly:

```
class Person {
    val name: String = MyJavaApi.getProperty("name")
}
```

A local value initialized with an expression of a platform type may or may not have a type declaration:

```
fun main() {
    val name = MyJavaApi.getProperty("name")
```

```
    println(name)
}
```

Scope functions apply/with/run/also/let

Kotlin provides a set of functions to execute a block of code in the context of a given object: let, run, with, apply, and also. For the guidance on choosing the right scope function for your case, refer to Scope Functions.

Coding conventions for libraries

When writing libraries, it's recommended to follow an additional set of rules to ensure API stability:

- Always explicitly specify member visibility (to avoid accidentally exposing declarations as public API)
- Always explicitly specify function return types and property types (to avoid accidentally changing the return type when the implementation changes)
- Provide KDoc comments for all public members, except for overrides that do not require any new documentation (to support generating documentation for the library)

Learn more about best practices and ideas to consider when writing an API for your library in library creators' guidelines.

Basic types

In Kotlin, everything is an object in the sense that you can call member functions and properties on any variable. Some types can have a special internal representation – for example, numbers, characters and booleans can be represented as primitive values at runtime – but to the user they look like ordinary classes.

This section describes the basic types used in Kotlin:

- Numbers and their unsigned counterparts
- Booleans
- Characters
- Strings
- Arrays

Numbers

Integer types

Kotlin provides a set of built-in types that represent numbers.
For integer numbers, there are four types with different sizes and, hence, value ranges:

Type	Size (bits)	Min value	Max value
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648 (-231)	2,147,483,647 (231 - 1)
Long	64	-9,223,372,036,854,775,808 (-263)	9,223,372,036,854,775,807 (263 - 1)

When you initialize a variable with no explicit type specification, the compiler automatically infers the type with the smallest range enough to represent the value. If it is not exceeding the range of Int, the type is Int. If it exceeds, the type is Long. To specify the Long value explicitly, append the suffix L to the value. Explicit type specification triggers the compiler to check the value not to exceed the range of the specified type.

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

> In addition to integer types, Kotlin also provides unsigned integer types. For more information, see Unsigned integer types.

Floating-point types

For real numbers, Kotlin provides floating-point types Float and Double that adhere to the IEEE 754 standard. Float reflects the IEEE 754 single precision, while Double reflects double precision.

These types differ in their size and provide storage for floating-point numbers with different precision:

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
Float	32	24	8	6-7
Double	64	53	11	15-16

You can initialize Double and Float variables with numbers having a fractional part. It's separated from the integer part by a period (.) For variables initialized with fractional numbers, the compiler infers the Double type:

```
val pi = 3.14 // Double
// val one: Double = 1 // Error: type mismatch
val oneDouble = 1.0 // Double
```

To explicitly specify the Float type for a value, add the suffix f or F. If such a value contains more than 6-7 decimal digits, it will be rounded:

```
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float, actual value is 2.7182817
```

Unlike some other languages, there are no implicit widening conversions for numbers in Kotlin. For example, a function with a Double parameter can be called only on Double values, but not Float, Int, or other numeric values:

```
fun main() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.0
    val f = 1.0f

    printDouble(d)
//    printDouble(i) // Error: Type mismatch
//    printDouble(f) // Error: Type mismatch
}
```

To convert numeric values to different types, use explicit conversions.

Literal constants for numbers

There are the following kinds of literal constants for integral values:

- Decimals: 123

* Longs are tagged by a capital L: 123L
* Hexadecimals: 0x0F
* Binaries: 0b00001011

> Octal literals are not supported in Kotlin.

Kotlin also supports a conventional notation for floating-point numbers:

* Doubles by default: 123.5, 123.5e10
* Floats are tagged by f or F: 123.5f

You can use underscores to make number constants more readable:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

> There are also special tags for unsigned integer literals.
> Read more about literals for unsigned integer types.

Numbers representation on the JVM

On the JVM platform, numbers are stored as primitive types: int, double, and so on. Exceptions are cases when you create a nullable number reference such as Int? or use generics. In these cases numbers are boxed in Java classes Integer, Double, and so on.

Nullable references to the same number can refer to different objects:

```
fun main() {
//sampleStart
    val a: Int = 100
    val boxedA: Int? = a
    val anotherBoxedA: Int? = a

    val b: Int = 10000
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b

    println(boxedA === anotherBoxedA) // true
    println(boxedB === anotherBoxedB) // false
//sampleEnd
}
```

All nullable references to a are actually the same object because of the memory optimization that JVM applies to Integers between -128 and 127. It doesn't apply to the b references, so they are different objects.

On the other hand, they are still equal:

```
fun main() {
//sampleStart
    val b: Int = 10000
    println(b == b) // Prints 'true'
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b
    println(boxedB == anotherBoxedB) // Prints 'true'
//sampleEnd
}
```

Explicit number conversions

Due to different representations, smaller types are not subtypes of bigger ones. If they were, we would have troubles of the following sort:

```
// Hypothetical code, does not actually compile:
val a: Int? = 1 // A boxed Int (java.lang.Integer)
val b: Long? = a // Implicit conversion yields a boxed Long (java.lang.Long)
print(b == a) // Surprise! This prints "false" as Long's equals() checks whether the other is Long as well
```

So equality would have been lost silently, not to mention identity.

As a consequence, smaller types are NOT implicitly converted to bigger types. This means that assigning a value of type Byte to an Int variable requires an explicit conversion:

```
fun main() {
//sampleStart
    val b: Byte = 1 // OK, literals are checked statically
    // val i: Int = b // ERROR
    val i1: Int = b.toInt()
//sampleEnd
}
```

All number types support conversions to other types:

- toByte(): Byte
- toShort(): Short
- toInt(): Int
- toLong(): Long
- toFloat(): Float
- toDouble(): Double

In many cases, there is no need for explicit conversions because the type is inferred from the context, and arithmetical operations are overloaded for appropriate conversions, for example:

```
val l = 1L + 3 // Long + Int => Long
```

Operations on numbers

Kotlin supports the standard set of arithmetical operations over numbers: +, -, *, /, %. They are declared as members of appropriate classes:

```
fun main() {
//sampleStart
    println(1 + 2)
    println(2_500_000_000L - 1L)
    println(3.14 * 2.71)
    println(10.0 / 3)
//sampleEnd
}
```

You can also override these operators for custom classes. See Operator overloading for details.

Division of integers

Division between integers numbers always returns an integer number. Any fractional part is discarded.

```
fun main() {
//sampleStart
    val x = 5 / 2
    //println(x == 2.5) // ERROR: Operator '==' cannot be applied to 'Int' and 'Double'
    println(x == 2)
//sampleEnd
}
```

This is true for a division between any two integer types:

```kotlin
fun main() {
//sampleStart
    val x = 5L / 2
    println(x == 2L)
//sampleEnd
}
```

To return a floating-point type, explicitly convert one of the arguments to a floating-point type:

```kotlin
fun main() {
//sampleStart
    val x = 5 / 2.toDouble()
    println(x == 2.5)
//sampleEnd
}
```

Bitwise operations

Kotlin provides a set of bitwise operations on integer numbers. They operate on the binary level directly with bits of the numbers' representation. Bitwise operations are represented by functions that can be called in infix form. They can be applied only to Int and Long:

```kotlin
val x = (1 shl 2) and 0x000FF000
```

Here is the complete list of bitwise operations:

- shl(bits) – signed shift left
- shr(bits) – signed shift right
- ushr(bits) – unsigned shift right
- and(bits) – bitwise AND
- or(bits) – bitwise OR
- xor(bits) – bitwise XOR
- inv() – bitwise inversion

Floating-point numbers comparison

The operations on floating-point numbers discussed in this section are:

- Equality checks: a == b and a != b
- Comparison operators: a < b, a > b, a <= b, a >= b
- Range instantiation and range checks: a..b, x in a..b, x !in a..b

When the operands a and b are statically known to be Float or Double or their nullable counterparts (the type is declared or inferred or is a result of a smart cast), the operations on the numbers and the range that they form follow the IEEE 754 Standard for Floating-Point Arithmetic.

However, to support generic use cases and provide total ordering, the behavior is different for operands that are not statically typed as floating-point numbers. For example, Any, Comparable<...>, or Collection<T> types. In this case, the operations use the equals and compareTo implementations for Float and Double. As a result:

- NaN is considered equal to itself
- NaN is considered greater than any other element including POSITIVE_INFINITY
- -0.0 is considered less than 0.0

Here is an example that shows the difference in behavior between operands statically typed as floating-point numbers (Double.NaN) and operands not statically typed as floating-point numbers (listOf(T)).

```kotlin
fun main() {
    //sampleStart
```

```kotlin
    println(Double.NaN == Double.NaN)                 // false
    println(listOf(Double.NaN) == listOf(Double.NaN)) // true

    println(0.0 == -0.0)                              // true
    println(listOf(0.0) == listOf(-0.0))              // false

    println(listOf(Double.NaN, Double.POSITIVE_INFINITY, 0.0, -0.0).sorted())
    // [-0.0, 0.0, Infinity, NaN]
    //sampleEnd
}
```

Unsigned integer types

In addition to integer types, Kotlin provides the following types for unsigned integer numbers:

- UByte: an unsigned 8-bit integer, ranges from 0 to 255
- UShort: an unsigned 16-bit integer, ranges from 0 to 65535
- UInt: an unsigned 32-bit integer, ranges from 0 to 2^32 - 1
- ULong: an unsigned 64-bit integer, ranges from 0 to 2^64 - 1

Unsigned types support most of the operations of their signed counterparts.

> Unsigned numbers are implemented as inline classes with the single storage property of the corresponding signed counterpart type of the same width. Nevertheless, changing type from unsigned type to signed counterpart (and vice versa) is a binary incompatible change.

Unsigned arrays and ranges

> Unsigned arrays and operations on them are in Beta. They can be changed incompatibly at any time. Opt-in is required (see the details below).

Same as for primitives, each of unsigned type has corresponding type that represents arrays of that type:

- UByteArray: an array of unsigned bytes
- UShortArray: an array of unsigned shorts
- UIntArray: an array of unsigned ints
- ULongArray: an array of unsigned longs

Same as for signed integer arrays, they provide similar API to Array class without boxing overhead.

When you use unsigned arrays, you'll get a warning that indicates that this feature is not stable yet. To remove the warning, opt-in the @ExperimentalUnsignedTypes annotation. It's up to you to decide if your clients have to explicitly opt-in into usage of your API, but keep in mind that unsigned arrays are not a stable feature, so API which uses them can be broken by changes in the language. Learn more about opt-in requirements.

Ranges and progressions are supported for UInt and ULong by classes UIntRange,UIntProgression, ULongRange, and ULongProgression. Together with the unsigned integer types, these classes are stable.

Unsigned integers literals

To make unsigned integers easier to use, Kotlin provides an ability to tag an integer literal with a suffix indicating a specific unsigned type (similarly to Float or Long):

- u and U tag is for unsigned literals. The exact type is determined based on the expected type. If no expected type is provided, compiler will use UInt or ULong depending on the size of literal:

  ```kotlin
  val b: UByte = 1u  // UByte, expected type provided
  val s: UShort = 1u // UShort, expected type provided
  ```

```kotlin
val l: ULong = 1u  // ULong, expected type provided

val a1 = 42u // UInt: no expected type provided, constant fits in UInt
val a2 = 0xFFFF_FFFF_FFFFu // ULong: no expected type provided, constant doesn't fit in UInt
```

- uL and UL explicitly tag literal as unsigned long:

```kotlin
val a = 1UL // ULong, even though no expected type provided and constant fits into UInt
```

Use cases

The main use case of unsigned numbers is utilizing the full bit range of an integer to represent positive values.
For example, to represent hexadecimal constants that do not fit in signed types such as color in 32-bit AARRGGBB format:

```kotlin
data class Color(val representation: UInt)

val yellow = Color(0xFFCC00CCu)
```

You can use unsigned numbers to initialize byte arrays without explicit toByte() literal casts:

```kotlin
val byteOrderMarkUtf8 = ubyteArrayOf(0xEFu, 0xBBu, 0xBFu)
```

Another use case is interoperability with native APIs. Kotlin allows representing native declarations that contain unsigned types in the signature. The mapping won't substitute unsigned integers with signed ones keeping the semantics unaltered.

Non-goals
While unsigned integers can only represent positive numbers and zero, it's not a goal to use them where application domain requires non-negative integers. For example, as a type of collection size or collection index value.

There are a couple of reasons:

- Using signed integers can help to detect accidental overflows and signal error conditions, such as List.lastIndex being -1 for an empty list.
- Unsigned integers cannot be treated as a range-limited version of signed ones because their range of values is not a subset of the signed integers range. Neither signed, nor unsigned integers are subtypes of each other.

Booleans

The type Boolean represents boolean objects that can have two values: true and false.

Boolean has a nullable counterpart Boolean? that also has the null value.

Built-in operations on booleans include:

- || – disjunction (logical OR)
- && – conjunction (logical AND)
- ! – negation (logical NOT)

|| and && work lazily.

```kotlin
fun main() {
//sampleStart
    val myTrue: Boolean = true
    val myFalse: Boolean = false
    val boolNull: Boolean? = null

    println(myTrue || myFalse)
    println(myTrue && myFalse)
    println(!myTrue)
//sampleEnd
}
```

On JVM: nullable references to boolean objects are boxed similarly to numbers.

Characters

Characters are represented by the type Char. Character literals go in single quotes: '1'.

Special characters start from an escaping backslash \. The following escape sequences are supported:

- \t – tab
- \b – backspace
- \n – new line (LF)
- \r – carriage return (CR)
- \' – single quotation mark
- \" – double quotation mark
- \\ – backslash
- \$ – dollar sign

To encode any other character, use the Unicode escape sequence syntax: '\uFF00'.

```
fun main() {
//sampleStart
    val aChar: Char = 'a'

    println(aChar)
    println('\n') // Prints an extra newline character
    println('\uFF00')
//sampleEnd
}
```

If a value of character variable is a digit, you can explicitly convert it to an Int number using the digitToInt() function.

On JVM: Like numbers, characters are boxed when a nullable reference is needed. Identity is not preserved by the boxing operation.

Strings

Strings in Kotlin are represented by the type String. Generally, a string value is a sequence of characters in double quotes ("):

```
val str = "abcd 123"
```

Elements of a string are characters that you can access via the indexing operation: s[i]. You can iterate over these characters with a for loop:

```
fun main() {
val str = "abcd"
//sampleStart
for (c in str) {
    println(c)
}
//sampleEnd
}
```

Strings are immutable. Once you initialize a string, you can't change its value or assign a new value to it. All operations that transform strings return their results in a new String object, leaving the original string unchanged:

```
fun main() {
//sampleStart
```

```
    val str = "abcd"
    println(str.uppercase()) // Create and print a new String object
    println(str) // The original string remains the same
//sampleEnd
}
```

To concatenate strings, use the + operator. This also works for concatenating strings with values of other types, as long as the first element in the expression is a string:

```
fun main() {
//sampleStart
val s = "abc" + 1
println(s + "def")
//sampleEnd
}
```

> In most cases using string templates or raw strings is preferable to string concatenation.

String literals

Kotlin has two types of string literals:

- Escaped strings
- Raw strings

Escaped strings

Escaped strings can contain escaped characters.
Here's an example of an escaped string:

```
val s = "Hello, world!\n"
```

Escaping is done in the conventional way, with a backslash (\).
See Characters page for the list of supported escape sequences.

Raw strings

Raw strings can contain newlines and arbitrary text. It is delimited by a triple quote ("""), contains no escaping and can contain newlines and any other characters:

```
val text = """
    for (c in "foo")
        print(c)
"""
```

To remove leading whitespace from raw strings, use the trimMargin() function:

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

By default, a pipe symbol | is used as margin prefix, but you can choose another character and pass it as a parameter, like trimMargin(">").

String templates

String literals may contain template expressions – pieces of code that are evaluated and whose results are concatenated into the string. A template expression starts with a dollar sign ($) and consists of either a name:

```kotlin
fun main() {
//sampleStart
    val i = 10
    println("i = $i") // Prints "i = 10"
//sampleEnd
}
```

or an expression in curly braces:

```kotlin
fun main() {
//sampleStart
    val s = "abc"
    println("$s.length is ${s.length}") // Prints "abc.length is 3"
//sampleEnd
}
```

You can use templates both in raw and escaped strings. To insert the dollar sign $ in a raw string (which doesn't support backslash escaping) before any symbol, which is allowed as a beginning of an identifier, use the following syntax:

```kotlin
val price = """
${'$'}_9.99
"""
```

Arrays

Arrays in Kotlin are represented by the Array class. It has get() and set() functions that turn into [] by operator overloading conventions, and the size property, along with other useful member functions:

```kotlin
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

To create an array, use the function arrayOf() and pass the item values to it, so that arrayOf(1, 2, 3) creates an array [1, 2, 3]. Alternatively, the arrayOfNulls() function can be used to create an array of a given size filled with null elements.

Another option is to use the Array constructor that takes the array size and the function that returns values of array elements given its index:

```kotlin
fun main() {
//sampleStart
    // Creates an Array<String> with values ["0", "1", "4", "9", "16"]
    val asc = Array(5) { i -> (i * i).toString() }
    asc.forEach { println(it) }
//sampleEnd
}
```

The [] operation stands for calls to member functions get() and set().

Arrays in Kotlin are invariant. This means that Kotlin does not let us assign an Array<String> to an Array<Any>, which prevents a possible runtime failure (but you can use Array<out Any>, see Type Projections).

Primitive type arrays

Kotlin also has classes that represent arrays of primitive types without boxing overhead: ByteArray, ShortArray, IntArray, and so on. These classes have no inheritance relation to the Array class, but they have the same set of methods and properties. Each of them also has a corresponding factory function:

```kotlin
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

```kotlin
// Array of int of size 5 with values [0, 0, 0, 0, 0]
val arr = IntArray(5)

// Example of initializing the values in the array with a constant
// Array of int of size 5 with values [42, 42, 42, 42, 42]
val arr = IntArray(5) { 42 }

// Example of initializing the values in the array using a lambda
// Array of int of size 5 with values [0, 1, 2, 3, 4] (values initialized to their index value)
var arr = IntArray(5) { it * 1 }
```

Type checks and casts

is and !is operators

Use the is operator or its negated form !is to perform a runtime check that identifies whether an object conforms to a given type:

```kotlin
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // same as !(obj is String)
    print("Not a String")
} else {
    print(obj.length)
}
```

Smart casts

In most cases, you don't need to use explicit cast operators in Kotlin because the compiler tracks the is-checks and explicit casts for immutable values and inserts (safe) casts automatically when necessary:

```kotlin
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

The compiler is smart enough to know that a cast is safe if a negative check leads to a return:

```kotlin
if (x !is String) return

print(x.length) // x is automatically cast to String
```

or if it is on the right-hand side of && or || and the proper check (regular or negative) is on the left-hand side:

```kotlin
// x is automatically cast to String on the right-hand side of `||`
if (x !is String || x.length == 0) return

// x is automatically cast to String on the right-hand side of `&&`
if (x is String && x.length > 0) {
    print(x.length) // x is automatically cast to String
}
```

Smart casts work for when expressions and while loops as well:

```kotlin
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

Note that smart casts work only when the compiler can guarantee that the variable won't change between the check and the usage. More specifically, smart casts

can be used under the following conditions:

- val local variables - always, with the exception of local delegated properties.
- val properties - if the property is private or internal or if the check is performed in the same module where the property is declared. Smart casts cannot be used on open properties or properties that have custom getters.
- var local variables - if the variable is not modified between the check and the usage, is not captured in a lambda that modifies it, and is not a local delegated property.
- var properties - never, because the variable can be modified at any time by other code.

"Unsafe" cast operator

Usually, the cast operator throws an exception if the cast isn't possible. And so, it's called unsafe. The unsafe cast in Kotlin is done by the infix operator as.

```
val x: String = y as String
```

Note that null cannot be cast to String, as this type is not nullable. If y is null, the code above throws an exception. To make code like this correct for null values, use the nullable type on the right-hand side of the cast:

```
val x: String? = y as String?
```

"Safe" (nullable) cast operator

To avoid exceptions, use the safe cast operator as?, which returns null on failure.

```
val x: String? = y as? String
```

Note that despite the fact that the right-hand side of as? is a non-null type String, the result of the cast is nullable.

Generics type checks and casts

Please see the corresponding section in the generics documentation page for information on which type checks and casts you can perform with generics.

Conditions and loops

If expression

In Kotlin, if is an expression: it returns a value. Therefore, there is no ternary operator (condition ? then : else) because ordinary if works fine in this role.

```
fun main() {
    val a = 2
    val b = 3

    //sampleStart
    var max = a
    if (a < b) max = b

    // With else
    if (a > b) {
      max = a
    } else {
      max = b
    }

    // As expression
    max = if (a > b) a else b

    // You can also use `else if` in expressions:
```

```kotlin
    val maxLimit = 1
    val maxOrLimit = if (maxLimit > a) maxLimit else if (a > b) a else b

    //sampleEnd
    println("max is $max")
    println("maxOrLimit is $maxOrLimit")
}
```

Branches of an if expression can be blocks. In this case, the last expression is the value of a block:

```kotlin
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

If you're using if as an expression, for example, for returning its value or assigning it to a variable, the else branch is mandatory.

When expression

when defines a conditional expression with multiple branches. It is similar to the switch statement in C-like languages. Its simple form looks like this.

```kotlin
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

when matches its argument against all branches sequentially until some branch condition is satisfied.

when can be used either as an expression or as a statement. If it is used as an expression, the value of the first matching branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. Just like with if, each branch can be a block, and its value is the value of the last expression in the block.

The else branch is evaluated if none of the other branch conditions are satisfied.

If when is used as an expression, the else branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions, for example, with enum class entries and sealed class subtypes).

```kotlin
enum class Bit {
    ZERO, ONE
}

val numericValue = when (getRandomBit()) {
    Bit.ZERO -> 0
    Bit.ONE -> 1
    // 'else' is not required because all cases are covered
}
```

In when statements, the else branch is mandatory in the following conditions:

- when has a subject of a Boolean, enum, or sealed type, or their nullable counterparts.
- branches of when don't cover all possible cases for this subject.

```kotlin
enum class Color {
    RED, GREEN, BLUE
}

when (getColor()) {
    Color.RED -> println("red")
    Color.GREEN -> println("green")
    Color.BLUE -> println("blue")
    // 'else' is not required because all cases are covered
}
```

```
when (getColor()) {
    Color.RED -> println("red") // no branches for GREEN and BLUE
    else -> println("not red") // 'else' is required
}
```

To define a common behavior for multiple cases, combine their conditions in a single line with a comma:

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

You can use arbitrary expressions (not only constants) as branch conditions

```
when (x) {
    s.toInt() -> print("s encodes x")
    else -> print("s does not encode x")
}
```

You can also check a value for being in or !in a range or a collection:

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

Another option is checking that a value is or !is of a particular type. Note that, due to smart casts, you can access the methods and properties of the type without any extra checks.

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

when can also be used as a replacement for an if-else if chain. If no argument is supplied, the branch conditions are simply boolean expressions, and a branch is executed when its condition is true:

```
when {
    x.isOdd() -> print("x is odd")
    y.isEven() -> print("y is even")
    else -> print("x+y is odd")
}
```

You can capture when subject in a variable using following syntax:

```
fun Request.getBody() =
    when (val response = executeRequest()) {
        is Success -> response.body
        is HttpError -> throw HttpException(response.status)
    }
```

The scope of variable introduced in when subject is restricted to the body of this when.

For loops

The for loop iterates through anything that provides an iterator. This is equivalent to the foreach loop in languages like C#. The syntax of for is the following:

```
for (item in collection) print(item)
```

The body of for can be a block.

```
for (item: Int in ints) {
```

```
    // ...
}
```

As mentioned before, for iterates through anything that provides an iterator. This means that it:

- has a member or an extension function iterator() that returns Iterator<>:
  - has a member or an extension function next()
  - has a member or an extension function hasNext() that returns Boolean.

All of these three functions need to be marked as operator.

To iterate over a range of numbers, use a range expression:

```
fun main() {
//sampleStart
    for (i in 1..3) {
        println(i)
    }
    for (i in 6 downTo 0 step 2) {
        println(i)
    }
//sampleEnd
}
```

A for loop over a range or an array is compiled to an index-based loop that does not create an iterator object.

If you want to iterate through an array or a list with an index, you can do it this way:

```
fun main() {
val array = arrayOf("a", "b", "c")
//sampleStart
    for (i in array.indices) {
        println(array[i])
    }
//sampleEnd
}
```

Alternatively, you can use the withIndex library function:

```
fun main() {
    val array = arrayOf("a", "b", "c")
//sampleStart
    for ((index, value) in array.withIndex()) {
        println("the element at $index is $value")
    }
//sampleEnd
}
```

While loops

while and do-while loops execute their body continuously while their condition is satisfied. The difference between them is the condition checking time:

- while checks the condition and, if it's satisfied, executes the body and then returns to the condition check.
- do-while executes the body and then checks the condition. If it's satisfied, the loop repeats. So, the body of do-while executes at least once regardless of the condition.

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

Break and continue in loops

Kotlin supports traditional break and continue operators in loops. See Returns and jumps.

Returns and jumps

Kotlin has three structural jump expressions:

- return by default returns from the nearest enclosing function or anonymous function.
- break terminates the nearest enclosing loop.
- continue proceeds to the next step of the nearest enclosing loop.

All of these expressions can be used as part of larger expressions:

```
val s = person.name ?: return
```

The type of these expressions is the Nothing type.

Break and continue labels

Any expression in Kotlin may be marked with a label. Labels have the form of an identifier followed by the @ sign, such as abc@ or fooBar@. To label an expression, just add a label in front of it.

```
loop@ for (i in 1..100) {
    // ...
}
```

Now, we can qualify a break or a continue with a label:

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

A break qualified with a label jumps to the execution point right after the loop marked with that label. A continue proceeds to the next iteration of that loop.

Return to labels

In Kotlin, functions can be nested using function literals, local functions, and object expressions. Qualified returns allow us to return from an outer function. The most important use case is returning from a lambda expression. Recall that when we write the following, the return-expression returns from the nearest enclosing function - foo:

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return // non-local return directly to the caller of foo()
        print(it)
    }
    println("this point is unreachable")
}
//sampleEnd

fun main() {
    foo()
}
```

Note that such non-local returns are supported only for lambda expressions passed to inline functions. To return from a lambda expression, label it and qualify the return:

```kotlin
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit // local return to the caller of the lambda - the forEach loop
        print(it)
    }
    print(" done with explicit label")
}
//sampleEnd

fun main() {
    foo()
}
```

Now, it returns only from the lambda expression. Often it is more convenient to use implicit labels, because such a label has the same name as the function to which the lambda is passed.

```kotlin
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return@forEach // local return to the caller of the lambda - the forEach loop
        print(it)
    }
    print(" done with implicit label")
}
//sampleEnd

fun main() {
    foo()
}
```

Alternatively, you can replace the lambda expression with an anonymous function. A return statement in an anonymous function will return from the anonymous function itself.

```kotlin
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach(fun(value: Int) {
        if (value == 3) return  // local return to the caller of the anonymous function - the forEach loop
        print(value)
    })
    print(" done with anonymous function")
}
//sampleEnd

fun main() {
    foo()
}
```

Note that the use of local returns in the previous three examples is similar to the use of continue in regular loops.

There is no direct equivalent for break, but it can be simulated by adding another nesting lambda and non-locally returning from it:

```kotlin
//sampleStart
fun foo() {
    run loop@{
        listOf(1, 2, 3, 4, 5).forEach {
            if (it == 3) return@loop // non-local return from the lambda passed to run
            print(it)
        }
    }
    print(" done with nested loop")
}
//sampleEnd

fun main() {
    foo()
}
```

When returning a value, the parser gives preference to the qualified return:

```kotlin
return@a 1
```

This means "return 1 at label @a" rather than "return a labeled expression (@a 1)".

Exceptions

Exception classes

All exception classes in Kotlin inherit the Throwable class. Every exception has a message, a stack trace, and an optional cause.

To throw an exception object, use the throw expression:

```
fun main() {
//sampleStart
    throw Exception("Hi There!")
//sampleEnd
}
```

To catch an exception, use the try...catch expression:

```
try {
    // some code
} catch (e: SomeException) {
    // handler
} finally {
    // optional finally block
}
```

There may be zero or more catch blocks, and the finally block may be omitted. However, at least one catch or finally block is required.

Try is an expression

try is an expression, which means it can have a return value:

```
val a: Int? = try { input.toInt() } catch (e: NumberFormatException) { null }
```

The returned value of a try expression is either the last expression in the try block or the last expression in the catch block (or blocks). The contents of the finally block don't affect the result of the expression.

Checked exceptions

Kotlin does not have checked exceptions. There are many reasons for this, but we will provide a simple example that illustrates why it is the case.

The following is an example interface from the JDK implemented by the StringBuilder class:

```
Appendable append(CharSequence csq) throws IOException;
```

This signature says that every time I append a string to something (a StringBuilder, some kind of a log, a console, etc.), I have to catch the IOExceptions. Why? Because the implementation might be performing IO operations (Writer also implements Appendable). The result is code like this all over the place:

```
try {
    log.append(message)
} catch (IOException e) {
    // Must be safe
}
```

And that's not good. Just take a look at Effective Java, 3rd Edition, Item 77: Don't ignore exceptions.

Bruce Eckel says this about checked exceptions:

> Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality.

And here are some additional thoughts on the matter:

- Java's checked exceptions were a mistake (Rod Waldhoff)
- The Trouble with Checked Exceptions (Anders Hejlsberg)

If you want to alert callers about possible exceptions when calling Kotlin code from Java, Swift, or Objective-C, you can use the @Throws annotation. Read more about using this annotation for Java and for Swift and Objective-C.

The Nothing type

throw is an expression in Kotlin, so you can use it, for example, as part of an Elvis expression:

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

The throw expression has the type Nothing. This type has no values and is used to mark code locations that can never be reached. In your own code, you can use Nothing to mark a function that never returns:

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

When you call this function, the compiler will know that the execution doesn't continue beyond the call:

```
val s = person.name ?: fail("Name required")
println(s)     // 's' is known to be initialized at this point
```

You may also encounter this type when dealing with type inference. The nullable variant of this type, Nothing?, has exactly one possible value, which is null. If you use null to initialize a value of an inferred type and there's no other information that can be used to determine a more specific type, the compiler will infer the Nothing? type:

```
val x = null           // 'x' has type `Nothing?`
val l = listOf(null)   // 'l' has type `List<Nothing?>
```

Java interoperability

Please see the section on exceptions in the Java interoperability page for information about Java interoperability.

Packages and imports

A source file may start with a package declaration:

```
package org.example

fun printMessage() { /*...*/ }
class Message { /*...*/ }

// ...
```

All the contents, such as classes and functions, of the source file are included in this package. So, in the example above, the full name of printMessage() is org.example.printMessage, and the full name of Message is org.example.Message.

If the package is not specified, the contents of such a file belong to the default package with no name.

Default imports

A number of packages are imported into every Kotlin file by default:

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.*
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

Additional packages are imported depending on the target platform:

- JVM:
  - java.lang.*
  - kotlin.jvm.*
- JS:
  - kotlin.js.*

Imports

Apart from the default imports, each file may contain its own import directives.

You can import either a single name:

```
import org.example.Message // Message is now accessible without qualification
```

or all the accessible contents of a scope: package, class, object, and so on:

```
import org.example.* // everything in 'org.example' becomes accessible
```

If there is a name clash, you can disambiguate by using as keyword to locally rename the clashing entity:

```
import org.example.Message // Message is accessible
import org.test.Message as TestMessage // TestMessage stands for 'org.test.Message'
```

The import keyword is not restricted to importing classes; you can also use it to import other declarations:

- top-level functions and properties
- functions and properties declared in object declarations
- enum constants

Visibility of top-level declarations

If a top-level declaration is marked private, it is private to the file it's declared in (see Visibility modifiers).

Classes

Classes in Kotlin are declared using the keyword class:

```
class Person { /*...*/ }
```

The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor, and some other things), and the class body surrounded by curly braces. Both the header and the body are optional; if the class has no body, the curly braces can be omitted.

```
class Empty
```

Constructors

A class in Kotlin can have a primary constructor and one or more secondary constructors. The primary constructor is a part of the class header, and it goes after the class name and optional type parameters.

```
class Person constructor(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted:

```
class Person(firstName: String) { /*...*/ }
```

The primary constructor cannot contain any code. Initialization code can be placed in initializer blocks prefixed with the init keyword.

During the initialization of an instance, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

```
//sampleStart
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
//sampleEnd

fun main() {
    InitOrderDemo("hello")
}
```

Primary constructor parameters can be used in the initializer blocks. They can also be used in property initializers declared in the class body:

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

Such declarations can also include default values of the class properties:

```
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = true)
```

You can use a trailing comma when you declare class properties:

```
class Person(
    val firstName: String,
    val lastName: String,
```

```
    var age: Int, // trailing comma
) { /*...*/ }
```

Much like regular properties, properties declared in the primary constructor can be mutable (var) or read-only (val).

If the constructor has annotations or visibility modifiers, the constructor keyword is required and the modifiers go before it:

```
class Customer public @Inject constructor(name: String) { /*...*/ }
```

Learn more about visibility modifiers.

Secondary constructors

A class can also declare secondary constructors, which are prefixed with constructor:

```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the this keyword:

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Code in initializer blocks effectively becomes part of the primary constructor. Delegation to the primary constructor happens at the moment of access to the first statement of a secondary constructor, so the code in all initializer blocks and property initializers is executed before the body of the secondary constructor.

Even if the class has no primary constructor, the delegation still happens implicitly, and the initializer blocks are still executed:

```
//sampleStart
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
//sampleEnd

fun main() {
    Constructors(1)
}
```

If a non-abstract class does not declare any constructors (primary or secondary), it will have a generated primary constructor with no arguments. The visibility of the constructor will be public.

If you don't want your class to have a public constructor, declare an empty primary constructor with non-default visibility:

```
class DontCreateMe private constructor() { /*...*/ }
```

On the JVM, if all of the primary constructor parameters have default values, the compiler will generate an additional parameterless constructor which will use the default values. This makes it easier to use Kotlin with libraries such as Jackson or JPA that create class instances through parameterless constructors.

```
class Customer(val customerName: String = "")
```

Creating instances of classes

To create an instance of a class, call the constructor as if it were a regular function:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

Kotlin does not have a new keyword.

The process of creating instances of nested, inner, and anonymous inner classes is described in Nested classes.

Class members

Classes can contain:

- Constructors and initializer blocks
- Functions
- Properties
- Nested and inner classes
- Object declarations

Inheritance

Classes can be derived from each other and form inheritance hierarchies. Learn more about inheritance in Kotlin.

Abstract classes

A class may be declared abstract, along with some or all of its members. An abstract member does not have an implementation in its class. You don't need to annotate abstract classes or functions with open.

```
abstract class Polygon {
    abstract fun draw()
}

class Rectangle : Polygon() {
    override fun draw() {
        // draw the rectangle
    }
}
```

You can override a non-abstract open member with an abstract one.

```
open class Polygon {
    open fun draw() {
        // some default polygon drawing method
    }
```

```
}

abstract class WildShape : Polygon() {
    // Classes that inherit WildShape need to provide their own
    // draw method instead of using the default on Polygon
    abstract override fun draw()
}
```

Companion objects

If you need to write a function that can be called without having a class instance but that needs access to the internals of a class (such as a factory method), you can write it as a member of an object declaration inside that class.

Even more specifically, if you declare a companion object inside your class, you can access its members using only the class name as a qualifier.

Inheritance

All classes in Kotlin have a common superclass, Any, which is the default superclass for a class with no supertypes declared:

```
class Example // Implicitly inherits from Any
```

Any has three methods: equals(), hashCode(), and toString(). Thus, these methods are defined for all Kotlin classes.

By default, Kotlin classes are final – they can't be inherited. To make a class inheritable, mark it with the open keyword:

```
open class Base // Class is open for inheritance
```

To declare an explicit supertype, place the type after a colon in the class header:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

If the derived class has a primary constructor, the base class can (and must) be initialized in that primary constructor according to its parameters.

If the derived class has no primary constructor, then each secondary constructor has to initialize the base type using the super keyword or it has to delegate to another constructor which does. Note that in this case different secondary constructors can call different constructors of the base type:

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Overriding methods

Kotlin requires explicit modifiers for overridable members and overrides:

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

The override modifier is required for Circle.draw(). If it's missing, the compiler will complain. If there is no open modifier on a function, like Shape.fill(), declaring a method with the same signature in a subclass is not allowed, either with override or without it. The open modifier has no effect when added to members of a final class – a class without an open modifier.

A member marked override is itself open, so it may be overridden in subclasses. If you want to prohibit re-overriding, use final:

```
open class Rectangle() : Shape() {
    final override fun draw() { /*...*/ }
}
```

Overriding properties

The overriding mechanism works on properties in the same way that it does on methods. Properties declared on a superclass that are then redeclared on a derived class must be prefaced with override, and they must have a compatible type. Each declared property can be overridden by a property with an initializer or by a property with a get method:

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
    override val vertexCount = 4
}
```

You can also override a val property with a var property, but not vice versa. This is allowed because a val property essentially declares a get method, and overriding it as a var additionally declares a set method in the derived class.

Note that you can use the override keyword as part of the property declaration in a primary constructor:

```
interface Shape {
    val vertexCount: Int
}

class Rectangle(override val vertexCount: Int = 4) : Shape // Always has 4 vertices

class Polygon : Shape {
    override var vertexCount: Int = 0  // Can be set to any number later
}
```

Derived class initialization order

During the construction of a new instance of a derived class, the base class initialization is done as the first step (preceded only by evaluation of the arguments for the base class constructor), which means that it happens before the initialization logic of the derived class is run.

```
//sampleStart
open class Base(val name: String) {

    init { println("Initializing a base class") }

    open val size: Int =
        name.length.also { println("Initializing size in the base class: $it") }
}

class Derived(
    name: String,
    val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also { println("Argument for the base class: $it") }) {

    init { println("Initializing a derived class") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in the derived class: $it") }
}
//sampleEnd

fun main() {
    println("Constructing the derived class(\"hello\", \"world\")")
    Derived("hello", "world")
}
```

This means that when the base class constructor is executed, the properties declared or overridden in the derived class have not yet been initialized. Using any of those properties in the base class initialization logic (either directly or indirectly through another overridden open member implementation) may lead to incorrect

behavior or a runtime failure. When designing a base class, you should therefore avoid using open members in the constructors, property initializers, or init blocks.

Calling the superclass implementation

Code in a derived class can call its superclass functions and property accessor implementations using the super keyword:

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

Inside an inner class, accessing the superclass of the outer class is done using the super keyword qualified with the outer class name: super@Outer:

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

//sampleStart
class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // Calls Rectangle's implementation of draw()
            fill()
            println("Drawn a filled rectangle with color ${super@FilledRectangle.borderColor}") // Uses Rectangle's implementation
of borderColor's get()
        }
    }
}
//sampleEnd

fun main() {
    val fr = FilledRectangle()
        fr.draw()
}
```

Overriding rules

In Kotlin, implementation inheritance is regulated by the following rule: if a class inherits multiple implementations of the same member from its immediate superclasses, it must override this member and provide its own implementation (perhaps, using one of the inherited ones).

To denote the supertype from which the inherited implementation is taken, use super qualified by the supertype name in angle brackets, such as super<Base>:

```
open class Rectangle {
    open fun draw() { /* ... */ }
}

interface Polygon {
    fun draw() { /* ... */ } // interface members are 'open' by default
}

class Square() : Rectangle(), Polygon {
    // The compiler requires draw() to be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
```

```
}
```

It's fine to inherit from both Rectangle and Polygon, but both of them have their implementations of draw(), so you need to override draw() in Square and provide a separate implementation for it to eliminate the ambiguity.

Properties

Declaring properties

Properties in Kotlin classes can be declared either as mutable, using the var keyword, or as read-only, using the val keyword.

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

To use a property, simply refer to it by its name:

```
fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}
```

Getters and setters

The full syntax for declaring a property is as follows:

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

The initializer, getter, and setter are optional. The property type is optional if it can be inferred from the initializer or the getter's return type, as shown below:

```
var initialized = 1 // has type Int, default getter and setter
// var allByDefault // ERROR: explicit initializer required, default getter and setter implied
```

The full syntax of a read-only property declaration differs from a mutable one in two ways: it starts with val instead of var and does not allow a setter:

```
val simple: Int? // has type Int, default getter, must be initialized in constructor
val inferredType = 1 // has type Int and a default getter
```

You can define custom accessors for a property. If you define a custom getter, it will be called every time you access the property (this way you can implement a computed property). Here's an example of a custom getter:

```
//sampleStart
class Rectangle(val width: Int, val height: Int) {
    val area: Int // property type is optional since it can be inferred from the getter's return type
        get() = this.width * this.height
}
//sampleEnd
fun main() {
    val rectangle = Rectangle(3, 4)
    println("Width=${rectangle.width}, height=${rectangle.height}, area=${rectangle.area}")
}
```

You can omit the property type if it can be inferred from the getter:

```
val area get() = this.width * this.height
```

If you define a custom setter, it will be called every time you assign a value to the property, except its initialization. A custom setter looks like this:

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // parses the string and assigns values to other properties
    }
```

By convention, the name of the setter parameter is value, but you can choose a different name if you prefer.

If you need to annotate an accessor or change its visibility, but you don't want to change the default implementation, you can define the accessor without defining its body:

```
var setterVisibility: String = "abc"
    private set // the setter is private and has the default implementation

var setterWithAnnotation: Any? = null
    @Inject set // annotate the setter with Inject
```

Backing fields

In Kotlin, a field is only used as a part of a property to hold its value in memory. Fields cannot be declared directly. However, when a property needs a backing field, Kotlin provides it automatically. This backing field can be referenced in the accessors using the field identifier:

```
var counter = 0 // the initializer assigns the backing field directly
    set(value) {
        if (value >= 0)
            field = value
            // counter = value // ERROR StackOverflow: Using actual name 'counter' would make setter recursive
    }
```

The field identifier can only be used in the accessors of the property.

A backing field will be generated for a property if it uses the default implementation of at least one of the accessors, or if a custom accessor references it through the field identifier.

For example, there would be no backing field in the following case:

```
val isEmpty: Boolean
    get() = this.size == 0
```

Backing properties

If you want to do something that does not fit into this implicit backing field scheme, you can always fall back to having a backing property:

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // Type parameters are inferred
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

On the JVM: Access to private properties with default getters and setters is optimized to avoid function call overhead.

Compile-time constants

If the value of a read-only property is known at compile time, mark it as a compile time constant using the const modifier. Such a property needs to fulfil the

following requirements:

- It must be a top-level property, or a member of an object declaration or a companion object.
- It must be initialized with a value of type String or a primitive type
- It cannot be a custom getter

The compiler will inline usages of the constant, replacing the reference to the constant with its actual value. However, the field will not be removed and therefore can be interacted with using reflection.

Such properties can also be used in annotations:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

Late-initialized properties and variables

Normally, properties declared as having a non-null type must be initialized in the constructor. However, it is often the case that doing so is not convenient. For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In these cases, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class.

To handle such cases, you can mark the property with the lateinit modifier:

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // dereference directly
    }
}
```

This modifier can be used on var properties declared inside the body of a class (not in the primary constructor, and only when the property does not have a custom getter or setter), as well as for top-level properties and local variables. The type of the property or variable must be non-null, and it must not be a primitive type.

Accessing a lateinit property before it has been initialized throws a special exception that clearly identifies the property being accessed and the fact that it hasn't been initialized.

Checking whether a lateinit var is initialized

To check whether a lateinit var has already been initialized, use .isInitialized on the reference to that property:

```
if (foo::bar.isInitialized) {
    println(foo.bar)
}
```

This check is only available for properties that are lexically accessible when declared in the same type, in one of the outer types, or at top level in the same file.

Overriding properties

See Overriding properties

Delegated properties

The most common kind of property simply reads from (and maybe writes to) a backing field, but custom getters and setters allow you to use properties so one can implement any sort of behavior of a property. Somewhere in between the simplicity of the first kind and variety of the second, there are common patterns for what properties can do. A few examples: lazy values, reading from a map by a given key, accessing a database, notifying a listener on access.

Such common behaviors can be implemented as libraries using delegated properties.

Interfaces

Interfaces in Kotlin can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store state. They can have properties, but these need to be abstract or provide accessor implementations.

An interface is defined using the keyword interface:

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

Implementing interfaces

A class or object can implement one or more interfaces:

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

Properties in interfaces

You can declare properties in interfaces. A property declared in an interface can either be abstract or provide implementations for accessors. Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them:

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

Interfaces Inheritance

An interface can derive from other interfaces, meaning it can both provide implementations for their members and declare new functions and properties. Quite naturally, classes implementing such an interface are only required to define the missing implementations:

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
```

```
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

Resolving overriding conflicts

When you declare many types in your supertype list, you may inherit more than one implementation of the same method:

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

Interfaces A and B both declare functions foo() and bar(). Both of them implement foo(), but only B implements bar() (bar() is not marked as abstract in A, because this is the default for interfaces if the function has no body). Now, if you derive a concrete class C from A, you have to override bar() and provide an implementation.

However, if you derive D from A and B, you need to implement all the methods that you have inherited from multiple interfaces, and you need to specify how exactly D should implement them. This rule applies both to methods for which you've inherited a single implementation (bar()) and to those for which you've inherited multiple implementations (foo()).

Functional (SAM) interfaces

An interface with only one abstract method is called a functional interface, or a Single Abstract Method (SAM) interface. The functional interface can have several non-abstract members but only one abstract member.

To declare a functional interface in Kotlin, use the fun modifier.

```
fun interface KRunnable {
   fun invoke()
}
```

SAM conversions

For functional interfaces, you can use SAM conversions that help make your code more concise and readable by using lambda expressions.

Instead of creating a class that implements a functional interface manually, you can use a lambda expression. With a SAM conversion, Kotlin can convert any lambda expression whose signature matches the signature of the interface's single method into the code, which dynamically instantiates the interface implementation.

For example, consider the following Kotlin functional interface:

```
fun interface IntPredicate {
   fun accept(i: Int): Boolean
```

```
}
```

If you don't use a SAM conversion, you will need to write code like this:

```
// Creating an instance of a class
val isEven = object : IntPredicate {
   override fun accept(i: Int): Boolean {
       return i % 2 == 0
   }
}
```

By leveraging Kotlin's SAM conversion, you can write the following equivalent code instead:

```
// Creating an instance using lambda
val isEven = IntPredicate { it % 2 == 0 }
```

A short lambda expression replaces all the unnecessary code.

```
fun interface IntPredicate {
   fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
   println("Is 7 even? - ${isEven.accept(7)}")
}
```

You can also use SAM conversions for Java interfaces.

Migration from an interface with constructor function to a functional interface

Starting from 1.6.20, Kotlin supports callable references to functional interface constructors, which adds a source-compatible way to migrate from an interface with a constructor function to a functional interface. Consider the following code:

```
interface Printer {
    fun print()
}

fun Printer(block: () -> Unit): Printer = object : Printer { override fun print() = block() }
```

With callable references to functional interface constructors enabled, this code can be replaced with just a functional interface declaration:

```
fun interface Printer {
    fun print()
}
```

Its constructor will be created implicitly, and any code using the ::Printer function reference will compile. For example:

```
documentsStorage.addPrinter(::Printer)
```

Preserve the binary compatibility by marking the legacy function Printer with the @Deprecated annotation with DeprecationLevel.HIDDEN:

```
@Deprecated(message = "Your message about the deprecation", level = DeprecationLevel.HIDDEN)
fun Printer(...) {...}
```

Functional interfaces vs. type aliases

You can also simply rewrite the above using a type alias for a functional type:

```
typealias IntPredicate = (i: Int) -> Boolean
```

```kotlin
val isEven: IntPredicate = { it % 2 == 0 }

fun main() {
   println("Is 7 even? - ${isEven(7)}")
}
```

However, functional interfaces and type aliases serve different purposes. Type aliases are just names for existing types – they don't create a new type, while functional interfaces do. You can provide extensions that are specific to a particular functional interface to be inapplicable for plain functions or their type aliases.

Type aliases can have only one member, while functional interfaces can have multiple non-abstract members and one abstract member. Functional interfaces can also implement and extend other interfaces.

Functional interfaces are more flexible and provide more capabilities than type aliases, but they can be more costly both syntactically and at runtime because they can require conversions to a specific interface. When you choose which one to use in your code, consider your needs:

- If your API needs to accept a function (any function) with some specific parameter and return types – use a simple functional type or define a type alias to give a shorter name to the corresponding functional type.
- If your API accepts a more complex entity than a function – for example, it has non-trivial contracts and/or operations on it that can't be expressed in a functional type's signature – declare a separate functional interface for it.

Visibility modifiers

Classes, objects, interfaces, constructors, and functions, as well as properties and their setters, can have visibility modifiers. Getters always have the same visibility as their properties.

There are four visibility modifiers in Kotlin: private, protected, internal, and public. The default visibility is public.

On this page, you'll learn how the modifiers apply to different types of declaring scopes.

Packages

Functions, properties, classes, objects, and interfaces can be declared at the "top-level" directly inside a package:

```kotlin
// file name: example.kt
package foo

fun baz() { ... }
class Bar { ... }
```

- If you don't use a visibility modifier, public is used by default, which means that your declarations will be visible everywhere.
- If you mark a declaration as private, it will only be visible inside the file that contains the declaration.
- If you mark it as internal, it will be visible everywhere in the same module.
- The protected modifier is not available for top-level declarations.

> To use a visible top-level declaration from another package, you should import it.

Examples:

```kotlin
// file name: example.kt
package foo

private fun foo() { ... } // visible inside example.kt

public var bar: Int = 5 // property is visible everywhere
    private set         // setter is visible only in example.kt

internal val baz = 6    // visible inside the same module
```

Class members

For members declared inside a class:

- private means that the member is visible inside this class only (including all its members).
- protected means that the member has the same visibility as one marked as private, but that it is also visible in subclasses.
- internal means that any client inside this module who sees the declaring class sees its internal members.
- public means that any client who sees the declaring class sees its public members.

> In Kotlin, an outer class does not see private members of its inner classes.

If you override a protected or an internal member and do not specify the visibility explicitly, the overriding member will also have the same visibility as the original.

Examples:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal open val c = 3
    val d = 4  // public by default

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a is not visible
    // b, c and d are visible
    // Nested and e are visible

    override val b = 5   // 'b' is protected
    override val c = 7   // 'c' is internal
}

class Unrelated(o: Outer) {
    // o.a, o.b are not visible
    // o.c and o.d are visible (same module)
    // Outer.Nested is not visible, and Nested::e is not visible either
}
```

Constructors

Use the following syntax to specify the visibility of the primary constructor of a class:

> You need to add an explicit constructor keyword.

```
class C private constructor(a: Int) { ... }
```

Here the constructor is private. By default, all constructors are public, which effectively amounts to them being visible everywhere the class is visible (this means that a constructor of an internal class is only visible within the same module).

Local declarations

Local variables, functions, and classes can't have visibility modifiers.

Modules

The internal visibility modifier means that the member is visible within the same module. More specifically, a module is a set of Kotlin files compiled together, for example:

- An IntelliJ IDEA module.
- A Maven project.
- A Gradle source set (with the exception that the test source set can access the internal declarations of main).
- A set of files compiled with one invocation of the <kotlinc> Ant task.

Extensions

Kotlin provides the ability to extend a class or an interface with new functionality without having to inherit from the class or use design patterns such as Decorator. This is done via special declarations called extensions.

For example, you can write new functions for a class or an interface from a third-party library that you can't modify. Such functions can be called in the usual way, as if they were methods of the original class. This mechanism is called an extension function. There are also extension properties that let you define new properties for existing classes.

Extension functions

To declare an extension function, prefix its name with a receiver type, which refers to the type being extended. The following adds a swap function to MutableList<Int>:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

The this keyword inside an extension function corresponds to the receiver object (the one that is passed before the dot). Now, you can call such a function on any MutableList<Int>:

```
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'list'
```

This function makes sense for any MutableList<T>, and you can make it generic:

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

You need to declare the generic type parameter before the function name to make it available in the receiver type expression. For more information about generics, see generic functions.

Extensions are resolved statically

Extensions do not actually modify the classes they extend. By defining an extension, you are not inserting new members into a class, only making new functions callable with the dot-notation on variables of this type.

Extension functions are dispatched statically, which means they are not virtual by receiver type. An extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result from evaluating that expression at runtime. For example:

```
fun main() {
//sampleStart
    open class Shape
    class Rectangle: Shape()

    fun Shape.getName() = "Shape"
    fun Rectangle.getName() = "Rectangle"

    fun printClassName(s: Shape) {
        println(s.getName())
```

```
    }

    printClassName(Rectangle())
//sampleEnd
}
```

This example prints Shape, because the extension function called depends only on the declared type of the parameter s, which is the Shape class.

If a class has a member function, and an extension function is defined which has the same receiver type, the same name, and is applicable to given arguments, the member always wins. For example:

```
fun main() {
//sampleStart
    class Example {
        fun printFunctionType() { println("Class method") }
    }

    fun Example.printFunctionType() { println("Extension function") }

    Example().printFunctionType()
//sampleEnd
}
```

This code prints Class method.

However, it's perfectly OK for extension functions to overload member functions that have the same name but a different signature:

```
fun main() {
//sampleStart
    class Example {
        fun printFunctionType() { println("Class method") }
    }

    fun Example.printFunctionType(i: Int) { println("Extension function #$i") }

    Example().printFunctionType(1)
//sampleEnd
}
```

Nullable receiver

Note that extensions can be defined with a nullable receiver type. These extensions can be called on an object variable even if its value is null, and they can check for this == null inside the body.

This way, you can call toString() in Kotlin without checking for null, as the check happens inside the extension function:

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // after the null check, 'this' is autocast to a non-null type, so the toString() below
    // resolves to the member function of the Any class
    return toString()
}
```

Extension properties

Kotlin supports extension properties much like it supports functions:

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

> Since extensions do not actually insert members into classes, there's no efficient way for an extension property to have a backing field. This is why initializers are not allowed for extension properties. Their behavior can only be defined by explicitly providing getters/setters.

Example:

```kotlin
val House.number = 1 // error: initializers are not allowed for extension properties
```

Companion object extensions

If a class has a companion object defined, you can also define extension functions and properties for the companion object. Just like regular members of the companion object, they can be called using only the class name as the qualifier:

```kotlin
class MyClass {
    companion object { }  // will be called "Companion"
}

fun MyClass.Companion.printCompanion() { println("companion") }

fun main() {
    MyClass.printCompanion()
}
```

Scope of extensions

In most cases, you define extensions on the top level, directly under packages:

```kotlin
package org.example.declarations

fun List<String>.getLongestString() { /*...*/}
```

To use an extension outside its declaring package, import it at the call site:

```kotlin
package org.example.usage

import org.example.declarations.getLongestString

fun main() {
    val list = listOf("red", "green", "blue")
    list.getLongestString()
}
```

See Imports for more information.

Declaring extensions as members

You can declare extensions for one class inside another class. Inside such an extension, there are multiple implicit receivers - objects whose members can be accessed without a qualifier. An instance of a class in which the extension is declared is called a dispatch receiver, and an instance of the receiver type of the extension method is called an extension receiver.

```kotlin
class Host(val hostname: String) {
    fun printHostname() { print(hostname) }
}

class Connection(val host: Host, val port: Int) {
    fun printPort() { print(port) }

    fun Host.printConnectionString() {
        printHostname()   // calls Host.printHostname()
        print(":")
        printPort()   // calls Connection.printPort()
    }

    fun connect() {
        /*...*/
        host.printConnectionString()   // calls the extension function
    }
}

fun main() {
    Connection(Host("kotl.in"), 443).connect()
```

```
    //Host("kotl.in").printConnectionString()  // error, the extension function is unavailable outside Connection
}
```

In the event of a name conflict between the members of a dispatch receiver and an extension receiver, the extension receiver takes precedence. To refer to the member of the dispatch receiver, you can use the qualified this syntax.

```
class Connection {
    fun Host.getConnectionString() {
        toString()         // calls Host.toString()
        this@Connection.toString()  // calls Connection.toString()
    }
}
```

Extensions declared as members can be declared as open and overridden in subclasses. This means that the dispatch of such functions is virtual with regard to the dispatch receiver type, but static with regard to the extension receiver type.

```
open class Base { }

class Derived : Base() { }

open class BaseCaller {
    open fun Base.printFunctionInfo() {
        println("Base extension function in BaseCaller")
    }

    open fun Derived.printFunctionInfo() {
        println("Derived extension function in BaseCaller")
    }

    fun call(b: Base) {
        b.printFunctionInfo()   // call the extension function
    }
}

class DerivedCaller: BaseCaller() {
    override fun Base.printFunctionInfo() {
        println("Base extension function in DerivedCaller")
    }

    override fun Derived.printFunctionInfo() {
        println("Derived extension function in DerivedCaller")
    }
}

fun main() {
    BaseCaller().call(Base())   // "Base extension function in BaseCaller"
    DerivedCaller().call(Base())  // "Base extension function in DerivedCaller" - dispatch receiver is resolved virtually
    DerivedCaller().call(Derived())  // "Base extension function in DerivedCaller" - extension receiver is resolved statically
}
```

Note on visibility

Extensions utilize the same visibility modifiers as regular functions declared in the same scope would. For example:

- An extension declared at the top level of a file has access to the other private top-level declarations in the same file.
- If an extension is declared outside its receiver type, it cannot access the receiver's private or protected members.

Data classes

It is not unusual to create classes whose main purpose is to hold data. In such classes, some standard functionality and some utility functions are often mechanically derivable from the data. In Kotlin, these are called data classes and are marked with data:

```
data class User(val name: String, val age: Int)
```

The compiler automatically derives the following members from all properties declared in the primary constructor:

- equals()/hashCode() pair

- toString() of the form "User(name=John, age=42)"
- componentN() functions corresponding to the properties in their order of declaration.
- copy() function (see below).

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:

- The primary constructor needs to have at least one parameter.
- All primary constructor parameters need to be marked as val or var.
- Data classes cannot be abstract, open, sealed, or inner.

Additionally, the generation of data class members follows these rules with regard to the members' inheritance:

- If there are explicit implementations of equals(), hashCode(), or toString() in the data class body or final implementations in a superclass, then these functions are not generated, and the existing implementations are used.
- If a supertype has componentN() functions that are open and return compatible types, the corresponding functions are generated for the data class and override those of the supertype. If the functions of the supertype cannot be overridden due to incompatible signatures or due to their being final, an error is reported.
- Providing explicit implementations for the componentN() and copy() functions is not allowed.

Data classes may extend other classes (see Sealed classes for examples).

On the JVM, if the generated class needs to have a parameterless constructor, default values for the properties have to be specified (see Constructors).

```
data class User(val name: String = "", val age: Int = 0)
```

Properties declared in the class body

The compiler only uses the properties defined inside the primary constructor for the automatically generated functions. To exclude a property from the generated implementations, declare it inside the class body:

```
data class Person(val name: String) {
    var age: Int = 0
}
```

Only the property name will be used inside the toString(), equals(), hashCode(), and copy() implementations, and there will only be one component function component1(). While two Person objects can have different ages, they will be treated as equal.

```
data class Person(val name: String) {
    var age: Int = 0
}
fun main() {
//sampleStart
    val person1 = Person("John")
    val person2 = Person("John")
    person1.age = 10
    person2.age = 20
//sampleEnd
    println("person1 == person2: ${person1 == person2}")
    println("person1 with age ${person1.age}: ${person1}")
    println("person2 with age ${person2.age}: ${person2}")
}
```

Copying

Use the copy() function to copy an object, allowing you to alter some of its properties while keeping the rest unchanged. The implementation of this function for the User class above would be as follows:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

You can then write the following:

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

Data classes and destructuring declarations

Component functions generated for data classes make it possible to use them in destructuring declarations:

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // prints "Jane, 35 years of age"
```

Standard data classes

The standard library provides the Pair and Triple classes. In most cases, though, named data classes are a better design choice because they make the code more readable by providing meaningful names for the properties.

Sealed classes and interfaces

Sealed classes and interfaces represent restricted class hierarchies that provide more control over inheritance. All direct subclasses of a sealed class are known at compile time. No other subclasses may appear outside the module and package within which the sealed class is defined. For example, third-party clients can't extend your sealed class in their code. Thus, each instance of a sealed class has a type from a limited set that is known when this class is compiled.

The same works for sealed interfaces and their implementations: once a module with a sealed interface is compiled, no new implementations can appear.

In some sense, sealed classes are similar to enum classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances, each with its own state.

As an example, consider a library's API. It's likely to contain error classes to let the library users handle errors that it can throw. If the hierarchy of such error classes includes interfaces or abstract classes visible in the public API, then nothing prevents implementing or extending them in the client code. However, the library doesn't know about errors declared outside it, so it can't treat them consistently with its own classes. With a sealed hierarchy of error classes, library authors can be sure that they know all possible error types and no other ones can appear later.

To declare a sealed class or interface, put the sealed modifier before its name:

```
sealed interface Error

sealed class IOError(): Error

class FileReadError(val file: File): IOError()
class DatabaseError(val source: DataSource): IOError()

object RuntimeError : Error
```

A sealed class is abstract by itself, it cannot be instantiated directly and can have abstract members.

Constructors of sealed classes can have one of two visibilities: protected (by default) or private:

```
sealed class IOError {
    constructor() { /*...*/ } // protected by default
    private constructor(description: String): this() { /*...*/ } // private is OK
    // public constructor(code: Int): this() {} // Error: public and internal are not allowed
}
```

Location of direct subclasses

Direct subclasses of sealed classes and interfaces must be declared in the same package. They may be top-level or nested inside any number of other named classes, named interfaces, or named objects. Subclasses can have any visibility as long as they are compatible with normal inheritance rules in Kotlin.

Subclasses of sealed classes must have a proper qualified name. They can't be local nor anonymous objects.

> enum classes can't extend a sealed class (as well as any other class), but they can implement sealed interfaces.

These restrictions don't apply to indirect subclasses. If a direct subclass of a sealed class is not marked as sealed, it can be extended in any way that its modifiers allow:

```
sealed interface Error // has implementations only in same package and module

sealed class IOError(): Error // extended only in same package and module
open class CustomError(): Error // can be extended wherever it's visible
```

Inheritance in multiplatform projects

There is one more inheritance restriction in multiplatform projects: direct subclasses of sealed classes must reside in the same source set. It applies to sealed classes without the expect and actual modifiers.

If a sealed class is declared as expect in a common source set and have actual implementations in platform source sets, both expect and actual versions can have subclasses in their source sets. Moreover, if you use a hierarchical structure, you can create subclasses in any source set between the expect and actual declarations.

Learn more about the hierarchical structure of multiplatform projects.

Sealed classes and when expression

The key benefit of using sealed classes comes into play when you use them in a when expression. If it's possible to verify that the statement covers all cases, you don't need to add an else clause to the statement:

```
fun log(e: Error) = when(e) {
    is FileReadError -> { println("Error while reading file ${e.file}") }
    is DatabaseError -> { println("Error while reading from database ${e.source}") }
    is RuntimeError ->  { println("Runtime error") }
    // the `else` clause is not required because all the cases are covered
}
```

> when expressions on expect sealed classes in the common code of multiplatform projects still require an else branch. This happens because subclasses of actual platform implementations aren't known in the common code.

Generics: in, out, where

Classes in Kotlin can have type parameters, just like in Java:

```
class Box<T>(t: T) {
    var value = t
}
```

To create an instance of such a class, simply provide the type arguments:

```
val box: Box<Int> = Box<Int>(1)
```

But if the parameters can be inferred, for example, from the constructor arguments, you can omit the type arguments:

```
val box = Box(1) // 1 has type Int, so the compiler figures out that it is Box<Int>
```

Variance

One of the trickiest aspects of Java's type system is the wildcard types (see Java Generics FAQ). Kotlin doesn't have these. Instead, Kotlin has declaration-site variance and type projections.

Let's think about why Java needs these mysterious wildcards. The problem is explained well in Effective Java, 3rd Edition, Item 31: Use bounded wildcards to increase API flexibility. First, generic types in Java are invariant, meaning that List<String> is not a subtype of List<Object>. If List were not invariant, it would have been no better than Java's arrays, as the following code would have compiled but caused an exception at runtime:

```
// Java
List<String> strs = new ArrayList<String>();
List<Object> objs = strs; // !!! A compile-time error here saves us from a runtime exception later.
objs.add(1); // Put an Integer into a list of Strings
String s = strs.get(0); // !!! ClassCastException: Cannot cast Integer to String
```

Java prohibits such things in order to guarantee run-time safety. But this has implications. For example, consider the addAll() method from the Collection interface. What's the signature of this method? Intuitively, you'd write it this way:

```
// Java
interface Collection<E> ... {
    void addAll(Collection<E> items);
}
```

But then, you would not be able to do the following (which is perfectly safe):

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from);
    // !!! Would not compile with the naive declaration of addAll:
    // Collection<String> is not a subtype of Collection<Object>
}
```

(In Java, you probably learned this the hard way, see Effective Java, 3rd Edition, Item 28: Prefer lists to arrays)

That's why the actual signature of addAll() is the following:

```
// Java
interface Collection<E> ... {
    void addAll(Collection<? extends E> items);
}
```

The wildcard type argument ? extends E indicates that this method accepts a collection of objects of E or a subtype of E, not just E itself. This means that you can safely read E's from items (elements of this collection are instances of a subclass of E), but cannot write to it as you don't know what objects comply with that unknown subtype of E. In return for this limitation, you get the desired behavior: Collection<String> is a subtype of Collection<? extends Object>. In other words, the wildcard with an extends-bound (upper bound) makes the type covariant.

The key to understanding why this works is rather simple: if you can only take items from a collection, then using a collection of Strings and reading Objects from it is fine. Conversely, if you can only put items into the collection, it's okay to take a collection of Objects and put Strings into it: in Java there is List<? super String>, a supertype of List<Object>.

The latter is called contravariance, and you can only call methods that take String as an argument on List<? super String> (for example, you can call add(String) or set(int, String)). If you call something that returns T in List<T>, you don't get a String, but rather an Object.

Joshua Bloch gives the name Producers to objects you only read from and Consumers to those you only write to. He recommends:

> "For maximum flexibility, use wildcard types on input parameters that represent producers or consumers", and proposes the following mnemonic:
>
> PECS stands for Producer-Extends, Consumer-Super.

> If you use a producer-object, say, List<? extends Foo>, you are not allowed to call add() or set() on this object, but this does not mean that it is immutable: for example, nothing prevents you from calling clear() to remove all the items from the list, since clear() does not take any parameters at all.
>
> The only thing guaranteed by wildcards (or other types of variance) is type safety. Immutability is a completely different story.

Declaration-site variance

Let's suppose that there is a generic interface Source<T> that does not have any methods that take T as a parameter, only methods that return T:

```
// Java
interface Source<T> {
    T nextT();
}
```

Then, it would be perfectly safe to store a reference to an instance of Source<String> in a variable of type Source<Object> - there are no consumer-methods to call. But Java does not know this, and still prohibits it:

```
// Java
void demo(Source<String> strs) {
    Source<Object> objects = strs; // !!! Not allowed in Java
    // ...
}
```

To fix this, you should declare objects of type Source<? extends Object>. Doing so is meaningless, because you can call all the same methods on such a variable as before, so there's no value added by the more complex type. But the compiler does not know that.

In Kotlin, there is a way to explain this sort of thing to the compiler. This is called declaration-site variance: you can annotate the type parameter T of Source to make sure that it is only returned (produced) from members of Source<T>, and never consumed. To do this, use the out modifier:

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
    // ...
}
```

The general rule is this: when a type parameter T of a class C is declared out, it may occur only in the out-position in the members of C, but in return C<Base> can safely be a supertype of C<Derived>.

In other words, you can say that the class C is covariant in the parameter T, or that T is a covariant type parameter. You can think of C as being a producer of T's, and NOT a consumer of T's.

The out modifier is called a variance annotation, and since it is provided at the type parameter declaration site, it provides declaration-site variance. This is in contrast with Java's use-site variance where wildcards in the type usages make the types covariant.

In addition to out, Kotlin provides a complementary variance annotation: in. It makes a type parameter contravariant, meaning it can only be consumed and never produced. A good example of a contravariant type is Comparable:

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, you can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

The words in and out seem to be self-explanatory (as they've already been used successfully in C# for quite some time), and so the mnemonic mentioned above is not really needed. It can in fact be rephrased at a higher level of abstraction:

The Existential Transformation: Consumer in, Producer out!:-)

Type projections

Use-site variance: type projections

It is very easy to declare a type parameter T as out and avoid trouble with subtyping on the use site, but some classes can't actually be restricted to only return T's! A good example of this is Array:

```kotlin
class Array<T>(val size: Int) {
    operator fun get(index: Int): T { ... }
    operator fun set(index: Int, value: T) { ... }
}
```

This class can be neither co- nor contravariant in T. And this imposes certain inflexibilities. Consider the following function:

```kotlin
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

This function is supposed to copy items from one array to another. Let's try to apply it in practice:

```kotlin
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any)
//   ^ type is Array<Int> but Array<Any> was expected
```

Here you run into the same familiar problem: Array<T> is invariant in T, and so neither Array<Int> nor Array<Any> is a subtype of the other. Why not? Again, this is because copy could have an unexpected behavior, for example, it may attempt to write a String to from, and if you actually pass an array of Int there, a ClassCastException will be thrown later.

To prohibit the copy function from writing to from, you can do the following:

```kotlin
fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

This is type projection, which means that from is not a simple array, but is rather a restricted (projected) one. You can only call methods that return the type parameter T, which in this case means that you can only call get(). This is our approach to use-site variance, and it corresponds to Java's Array<? extends Object> while being slightly simpler.

You can project a type with in as well:

```kotlin
fun fill(dest: Array<in String>, value: String) { ... }
```

Array<in String> corresponds to Java's Array<? super String>. This means that you can pass an array of CharSequence or an array of Object to the fill() function.

Star-projections

Sometimes you want to say that you know nothing about the type argument, but you still want to use it in a safe way. The safe way here is to define such a projection of the generic type, that every concrete instantiation of that generic type will be a subtype of that projection.

Kotlin provides so-called star-projection syntax for this:

- For Foo<out T : TUpper>, where T is a covariant type parameter with the upper bound TUpper, Foo<*> is equivalent to Foo<out TUpper>. This means that when the T is unknown you can safely read values of TUpper from Foo<*>.
- For Foo<in T>, where T is a contravariant type parameter, Foo<*> is equivalent to Foo<in Nothing>. This means there is nothing you can write to Foo<*> in a safe way when T is unknown.
- For Foo<T : TUpper>, where T is an invariant type parameter with the upper bound TUpper, Foo<*> is equivalent to Foo<out TUpper> for reading values and to Foo<in Nothing> for writing values.

If a generic type has several type parameters, each of them can be projected independently. For example, if the type is declared as interface Function<in T, out U> you could use the following star-projections:

- Function<*, String> means Function<in Nothing, String>.
- Function<Int, *> means Function<Int, out Any?>.
- Function<*, *> means Function<in Nothing, out Any?>.

> Star-projections are very much like Java's raw types, but safe.

Generic functions

Classes aren't the only declarations that can have type parameters. Functions can, too. Type parameters are placed before the name of the function:

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString(): String { // extension function
    // ...
}
```

To call a generic function, specify the type arguments at the call site after the name of the function:

```
val l = singletonList<Int>(1)
```

Type arguments can be omitted if they can be inferred from the context, so the following example works as well:

```
val l = singletonList(1)
```

Generic constraints

The set of all possible types that can be substituted for a given type parameter may be restricted by generic constraints.

Upper bounds

The most common type of constraint is an upper bound, which corresponds to Java's extends keyword:

```
fun <T : Comparable<T>> sort(list: List<T>) {  ... }
```

The type specified after a colon is the upper bound, indicating that only a subtype of Comparable<T> can be substituted for T. For example:

```
sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a subtype of Comparable<HashMap<Int, String>>
```

The default upper bound (if there was none specified) is Any?. Only one upper bound can be specified inside the angle brackets. If the same type parameter needs more than one upper bound, you need a separate where-clause:

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
    where T : CharSequence,
          T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```

The passed type must satisfy all conditions of the where clause simultaneously. In the above example, the T type must implement both CharSequence and Comparable.

Type erasure

The type safety checks that Kotlin performs for generic declaration usages are done at compile time. At runtime, the instances of generic types do not hold any information about their actual type arguments. The type information is said to be erased. For example, the instances of Foo<Bar> and Foo<Baz?> are erased to just Foo<*>.

Generics type checks and casts

Due to the type erasure, there is no general way to check whether an instance of a generic type was created with certain type arguments at runtime, and the compiler prohibits such is-checks such as ints is List<Int> or list is T (type parameter). However, you can check an instance against a star-projected type:

```
if (something is List<*>) {
    something.forEach { println(it) } // The items are typed as `Any?`
```

```
}
```

Similarly, when you already have the type arguments of an instance checked statically (at compile time), you can make an is-check or a cast that involves the non-generic part of the type. Note that angle brackets are omitted in this case:

```
fun handleStrings(list: MutableList<String>) {
    if (list is ArrayList) {
        // `list` is smart-cast to `ArrayList<String>`
    }
}
```

The same syntax but with the type arguments omitted can be used for casts that do not take type arguments into account: list as ArrayList.

The type arguments of a generic function calls are also only checked at compile time. Inside the function bodies, the type parameters cannot be used for type checks, and type casts to type parameters (foo as T) are unchecked. The only exclusion is inline functions with reified type parameters, which have their actual type arguments inlined at each call site. This enables type checks and casts for the type parameters. However, the restrictions described above still apply for instances of generic types used inside checks or casts. For example, in the type check arg is T, if arg is an instance of a generic type itself, its type arguments are still erased.

```
//sampleStart
inline fun <reified A, reified B> Pair<*, *>.asPairOf(): Pair<A, B>? {
    if (first !is A || second !is B) return null
    return first as A to second as B
}

val somePair: Pair<Any?, Any?> = "items" to listOf(1, 2, 3)


val stringToSomething = somePair.asPairOf<String, Any>()
val stringToInt = somePair.asPairOf<String, Int>()
val stringToList = somePair.asPairOf<String, List<*>>()
val stringToStringList = somePair.asPairOf<String, List<String>>() // Compiles but breaks type safety!
// Expand the sample for more details

//sampleEnd

fun main() {
    println("stringToSomething = " + stringToSomething)
    println("stringToInt = " + stringToInt)
    println("stringToList = " + stringToList)
    println("stringToStringList = " + stringToStringList)
    //println(stringToStringList?.second?.forEach() {it.length}) // This will throw ClassCastException as list items are not String
}
```

Unchecked casts

Type casts to generic types with concrete type arguments such as foo as List<String> cannot be checked at runtime.
These unchecked casts can be used when type safety is implied by the high-level program logic but cannot be inferred directly by the compiler. See the example below.

```
fun readDictionary(file: File): Map<String, *> = file.inputStream().use {
    TODO("Read a mapping of strings to arbitrary elements.")
}

// We saved a map with `Int`s into this file
val intsFile = File("ints.dictionary")

// Warning: Unchecked cast: `Map<String, *>` to `Map<String, Int>`
val intsDictionary: Map<String, Int> = readDictionary(intsFile) as Map<String, Int>
```

A warning appears for the cast in the last line. The compiler can't fully check it at runtime and provides no guarantee that the values in the map are Int.

To avoid unchecked casts, you can redesign the program structure. In the example above, you could use the DictionaryReader<T> and DictionaryWriter<T> interfaces with type-safe implementations for different types. You can introduce reasonable abstractions to move unchecked casts from the call site to the implementation details. Proper use of generic variance can also help.

For generic functions, using reified type parameters makes casts like arg as T checked, unless arg's type has its own type arguments that are erased.

An unchecked cast warning can be suppressed by annotating the statement or the declaration where it occurs with @Suppress("UNCHECKED_CAST"):

```
inline fun <reified T> List<*>.asListOfType(): List<T>? =
```

```kotlin
    if (all { it is T })
        @Suppress("UNCHECKED_CAST")
        this as List<T> else
        null
```

On the JVM: array types (Array<Foo>) retain information about the erased type of their elements, and type casts to an array type are partially checked: the nullability and actual type arguments of the element type are still erased. For example, the cast foo as Array<List<String>?> will succeed if foo is an array holding any List<*>, whether it is nullable or not.

Underscore operator for type arguments

The underscore operator _ can be used for type arguments. Use it to automatically infer a type of the argument when other types are explicitly specified:

```kotlin
abstract class SomeClass<T> {
    abstract fun execute() : T
}

class SomeImplementation : SomeClass<String>() {
    override fun execute(): String = "Test"
}

class OtherImplementation : SomeClass<Int>() {
    override fun execute(): Int = 42
}

object Runner {
    inline fun <reified S: SomeClass<T>, T> run() : T {
        return S::class.java.getDeclaredConstructor().newInstance().execute()
    }
}

fun main() {
    // T is inferred as String because SomeImplementation derives from SomeClass<String>
    val s = Runner.run<SomeImplementation, _>()
    assert(s == "Test")

    // T is inferred as Int because OtherImplementation derives from SomeClass<Int>
    val n = Runner.run<OtherImplementation, _>()
    assert(n == 42)
}
```

Nested and inner classes

Classes can be nested in other classes:

```kotlin
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

You can also use interfaces with nesting. All combinations of classes and interfaces are possible: You can nest interfaces in classes, classes in interfaces, and interfaces in interfaces.

```kotlin
interface OuterInterface {
    class InnerClass
    interface InnerInterface
}

class OuterClass {
    class InnerClass
    interface InnerInterface
}
```

Inner classes

A nested class marked as inner can access the members of its outer class. Inner classes carry a reference to an object of an outer class:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

See Qualified this expressions to learn about disambiguation of this in inner classes.

Anonymous inner classes

Anonymous inner class instances are created using an object expression:

```
window.addMouseListener(object : MouseAdapter() {

    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
})
```

On the JVM, if the object is an instance of a functional Java interface (that means a Java interface with a single abstract method), you can create it using a lambda expression prefixed with the type of the interface:

```
val listener = ActionListener { println("clicked") }
```

Enum classes

The most basic use case for enum classes is the implementation of type-safe enums:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

Each enum constant is an object. Enum constants are separated by commas.

Since each enum is an instance of the enum class, it can be initialized as:

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Anonymous classes

Enum constants can declare their own anonymous classes with their corresponding methods, as well as with overriding base methods.

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
```

```
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

If the enum class defines any members, separate the constant definitions from the member definitions with a semicolon.

Implementing interfaces in enum classes

An enum class can implement an interface (but it cannot derive from a class), providing either a common implementation of interface members for all the entries, or separate implementations for each entry within its anonymous class. This is done by adding the interfaces you want to implement to the enum class declaration as follows:

```
import java.util.function.BinaryOperator
import java.util.function.IntBinaryOperator

//sampleStart
enum class IntArithmetics : BinaryOperator<Int>, IntBinaryOperator {
    PLUS {
        override fun apply(t: Int, u: Int): Int = t + u
    },
    TIMES {
        override fun apply(t: Int, u: Int): Int = t * u
    };

    override fun applyAsInt(t: Int, u: Int) = apply(t, u)
}
//sampleEnd

fun main() {
    val a = 13
    val b = 31
    for (f in IntArithmetics.values()) {
        println("$f($a, $b) = ${f.apply(a, b)}")
    }
}
```

All enum classes implement the Comparable interface by default. Constants in the enum class are defined in the natural order. For more information, see Ordering.

Working with enum constants

Enum classes in Kotlin have synthetic methods for listing the defined enum constants and getting an enum constant by its name. The signatures of these methods are as follows (assuming the name of the enum class is EnumClass):

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

Below is an example of these methods in action:

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    for (color in RGB.values()) println(color.toString()) // prints RED, GREEN, BLUE
    println("The first color is: ${RGB.valueOf("RED")}") // prints "The first color is: RED"
}
```

The valueOf() method throws an IllegalArgumentException if the specified name does not match any of the enum constants defined in the class.

You can access the constants in an enum class in a generic way using the enumValues<T>() and enumValueOf<T>() functions:

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}

printAllValues<RGB>() // prints RED, GREEN, BLUE
```

For more information about inline functions and reified type parameters, see Inline functions.

In Kotlin 1.9.0, the entries property is introduced as a replacement for the values() function. The entries property returns a pre-allocated immutable list of your enum constants. This is particularly useful when you are working with collections and can you help you avoid performance issues.

For example:

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    for (color in RGB.entries) println(color.toString())
    // prints RED, GREEN, BLUE
}
```

Every enum constant also has properties: name and ordinal, for obtaining its name and position (starting from 0) in the enum class declaration:

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    //sampleStart
    println(RGB.RED.name) // prints RED
    println(RGB.RED.ordinal) // prints 0
    //sampleEnd
}
```

Inline value classes

Sometimes it is necessary for business logic to create a wrapper around some type. However, it introduces runtime overhead due to additional heap allocations. Moreover, if the wrapped type is primitive, the performance hit is terrible, because primitive types are usually heavily optimized by the runtime, while their wrappers don't get any special treatment.

To solve such issues, Kotlin introduces a special kind of class called an inline class. Inline classes are a subset of value-based classes. They don't have an identity and can only hold values.

To declare an inline class, use the value modifier before the name of the class:

```
value class Password(private val s: String)
```

To declare an inline class for the JVM backend, use the value modifier along with the @JvmInline annotation before the class declaration:

```
// For JVM backends
@JvmInline
value class Password(private val s: String)
```

An inline class must have a single property initialized in the primary constructor. At runtime, instances of the inline class will be represented using this single property (see details about runtime representation below):

```
// No actual instantiation of class 'Password' happens
// At runtime 'securePassword' contains just 'String'
val securePassword = Password("Don't try this in production")
```

This is the main feature of inline classes, which inspired the name inline: data of the class is inlined into its usages (similar to how content of inline functions is inlined to call sites).

Members

Inline classes support some functionality of regular classes. In particular, they are allowed to declare properties and functions, have an init block and secondary constructors:

```kotlin
@JvmInline
value class Person(private val fullName: String) {
    init {
        require(fullName.isNotEmpty()) {
            "Full name shouldn't be empty"
        }
    }

    constructor(firstName: String, lastName: String) : this("$firstName $lastName") {
        require(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }

    val length: Int
        get() = fullName.length

    fun greet() {
        println("Hello, $fullName")
    }
}

fun main() {
    val name1 = Person("Kotlin", "Mascot")
    val name2 = Person("Kodee")
    name1.greet() // the `greet()` function is called as a static method
    println(name2.length) // property getter is called as a static method
}
```

Inline class properties cannot have backing fields. They can only have simple computable properties (no lateinit/delegated properties).

Inheritance

Inline classes are allowed to inherit from interfaces:

```kotlin
interface Printable {
    fun prettyPrint(): String
}

@JvmInline
value class Name(val s: String) : Printable {
    override fun prettyPrint(): String = "Let's $s!"
}

fun main() {
    val name = Name("Kotlin")
    println(name.prettyPrint()) // Still called as a static method
}
```

It is forbidden for inline classes to participate in a class hierarchy. This means that inline classes cannot extend other classes and are always final.

Representation

In generated code, the Kotlin compiler keeps a wrapper for each inline class. Inline class instances can be represented at runtime either as wrappers or as the underlying type. This is similar to how Int can be represented either as a primitive int or as the wrapper Integer.

The Kotlin compiler will prefer using underlying types instead of wrappers to produce the most performant and optimized code. However, sometimes it is necessary to keep wrappers around. As a rule of thumb, inline classes are boxed whenever they are used as another type.

```kotlin
interface I

@JvmInline
value class Foo(val i: Int) : I

fun asInline(f: Foo) {}
fun <T> asGeneric(x: T) {}
fun asInterface(i: I) {}
fun asNullable(i: Foo?) {}

fun <T> id(x: T): T = x

fun main() {
```

```kotlin
    val f = Foo(42)

    asInline(f)    // unboxed: used as Foo itself
    asGeneric(f)   // boxed: used as generic type T
    asInterface(f) // boxed: used as type I
    asNullable(f)  // boxed: used as Foo?, which is different from Foo

    // below, 'f' first is boxed (while being passed to 'id') and then unboxed (when returned from 'id')
    // In the end, 'c' contains unboxed representation (just '42'), as 'f'
    val c = id(f)
}
```

Because inline classes may be represented both as the underlying value and as a wrapper, referential equality is pointless for them and is therefore prohibited.

Inline classes can also have a generic type parameter as the underlying type. In this case, the compiler maps it to Any? or, generally, to the upper bound of the type parameter.

```kotlin
@JvmInline
value class UserId<T>(val value: T)

fun compute(s: UserId<String>) {} // compiler generates fun compute-<hashcode>(s: Any?)
```

Mangling

Since inline classes are compiled to their underlying type, it may lead to various obscure errors, for example unexpected platform signature clashes:

```kotlin
@JvmInline
value class UInt(val x: Int)

// Represented as 'public final void compute(int x)' on the JVM
fun compute(x: Int) { }

// Also represented as 'public final void compute(int x)' on the JVM!
fun compute(x: UInt) { }
```

To mitigate such issues, functions using inline classes are mangled by adding some stable hashcode to the function name. Therefore, fun compute(x: UInt) will be represented as public final void compute-<hashcode>(int x), which solves the clash problem.

Calling from Java code

You can call functions that accept inline classes from Java code. To do so, you should manually disable mangling: add the @JvmName annotation before the function declaration:

```kotlin
@JvmInline
value class UInt(val x: Int)

fun compute(x: Int) { }

@JvmName("computeUInt")
fun compute(x: UInt) { }
```

Inline classes vs type aliases

At first sight, inline classes seem very similar to type aliases. Indeed, both seem to introduce a new type and both will be represented as the underlying type at runtime.

However, the crucial difference is that type aliases are assignment-compatible with their underlying type (and with other type aliases with the same underlying type), while inline classes are not.

In other words, inline classes introduce a truly new type, contrary to type aliases which only introduce an alternative name (alias) for an existing type:

```kotlin
typealias NameTypeAlias = String

@JvmInline
value class NameInlineClass(val s: String)

fun acceptString(s: String) {}
```

```kotlin
fun acceptNameTypeAlias(n: NameTypeAlias) {}
fun acceptNameInlineClass(p: NameInlineClass) {}

fun main() {
    val nameAlias: NameTypeAlias = ""
    val nameInlineClass: NameInlineClass = NameInlineClass("")
    val string: String = ""

    acceptString(nameAlias) // OK: pass alias instead of underlying type
    acceptString(nameInlineClass) // Not OK: can't pass inline class instead of underlying type

    // And vice versa:
    acceptNameTypeAlias(string) // OK: pass underlying type instead of alias
    acceptNameInlineClass(string) // Not OK: can't pass underlying type instead of inline class
}
```

Inline classes and delegation

Implementation by delegation to inlined value of inlined class is allowed with interfaces:

```kotlin
interface MyInterface {
    fun bar()
    fun foo() = "foo"
}

@JvmInline
value class MyInterfaceWrapper(val myInterface: MyInterface) : MyInterface by myInterface

fun main() {
    val my = MyInterfaceWrapper(object : MyInterface {
        override fun bar() {
            // body
        }
    })
    println(my.foo()) // prints "foo"
}
```

Object expressions and declarations

Sometimes you need to create an object that is a slight modification of some class, without explicitly declaring a new subclass for it. Kotlin can handle this with object expressions and object declarations.

Object expressions

Object expressions create objects of anonymous classes, that is, classes that aren't explicitly declared with the class declaration. Such classes are useful for one-time use. You can define them from scratch, inherit from existing classes, or implement interfaces. Instances of anonymous classes are also called anonymous objects because they are defined by an expression, not a name.

Creating anonymous objects from scratch

Object expressions start with the object keyword.

If you just need an object that doesn't have any nontrivial supertypes, write its members in curly braces after object:

```kotlin
fun main() {
//sampleStart
    val helloWorld = object {
        val hello = "Hello"
        val world = "World"
        // object expressions extend Any, so `override` is required on `toString()`
        override fun toString() = "$hello $world"
    }
//sampleEnd
    print(helloWorld)
}
```

Inheriting anonymous objects from supertypes

To create an object of an anonymous class that inherits from some type (or types), specify this type after object and a colon (:). Then implement or override the members of this class as if you were inheriting from it:

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { /*...*/ }

    override fun mouseEntered(e: MouseEvent) { /*...*/ }
})
```

If a supertype has a constructor, pass appropriate constructor parameters to it. Multiple supertypes can be specified as a comma-delimited list after the colon:

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { /*...*/ }

val ab: A = object : A(1), B {
    override val y = 15
}
```

Using anonymous objects as return and value types

When an anonymous object is used as a type of a local or private but not inline declaration (function or property), all its members are accessible via this function or property:

```
class C {
    private fun getObject() = object {
        val x: String = "x"
    }

    fun printX() {
        println(getObject().x)
    }
}
```

If this function or property is public or private inline, its actual type is:

- Any if the anonymous object doesn't have a declared supertype
- The declared supertype of the anonymous object, if there is exactly one such type
- The explicitly declared type if there is more than one declared supertype

In all these cases, members added in the anonymous object are not accessible. Overridden members are accessible if they are declared in the actual type of the function or property:

```
interface A {
    fun funFromA() {}
}
interface B

class C {
    // The return type is Any; x is not accessible
    fun getObject() = object {
        val x: String = "x"
    }

    // The return type is A; x is not accessible
    fun getObjectA() = object: A {
        override fun funFromA() {}
        val x: String = "x"
    }

    // The return type is B; funFromA() and x are not accessible
    fun getObjectB(): B = object: A, B { // explicit return type is required
        override fun funFromA() {}
        val x: String = "x"
    }
}
```

Accessing variables from anonymous objects

The code in object expressions can access variables from the enclosing scope:

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}
```

Object declarations

The Singleton pattern can be useful in several cases, and Kotlin makes it easy to declare singletons:

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

This is called an object declaration, and it always has a name following the object keyword. Just like a variable declaration, an object declaration is not an expression, and it cannot be used on the right-hand side of an assignment statement.

The initialization of an object declaration is thread-safe and done on first access.

To refer to the object, use its name directly:

```
DataProviderManager.registerDataProvider(...)
```

Such objects can have supertypes:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
}
```

> Object declarations can't be local (that is, they can't be nested directly inside a function), but they can be nested into other object declarations or non-inner classes.

Data objects

When printing a plain object declaration in Kotlin, the string representation contains both its name and the hash of the object:

```
object MyObject

fun main() {
    println(MyObject) // MyObject@1f32e575
}
```

Just like data classes, you can mark an object declaration with the data modifier. This instructs the compiler to generate a number of functions for your object:

- toString() returns the name of the data object

- equals()/hashCode() pair

> You can't provide a custom equals or hashCode implementation for a data object.

The toString() function of a data object returns the name of the object:

```
data object MyDataObject {
    val x: Int = 3
}

fun main() {
    println(MyDataObject) // MyDataObject
}
```

The equals() function for a data object ensures that all objects that have the type of your data object are considered equal. In most cases, you will only have a single instance of your data object at runtime (after all, a data object declares a singleton). However, in the edge case where another object of the same type is generated at runtime (for example, by using platform reflection with java.lang.reflect or a JVM serialization library that uses this API under the hood), this ensures that the objects are treated as being equal.

> Make sure that you only compare data objects structurally (using the == operator) and never by reference (using the === operator). This helps you to avoid pitfalls when more than one instance of a data object exists at runtime.

```
import java.lang.reflect.Constructor

data object MySingleton

fun main() {
    val evilTwin = createInstanceViaReflection()

    println(MySingleton) // MySingleton
    println(evilTwin) // MySingleton

    // Even when a library forcefully creates a second instance of MySingleton, its `equals` method returns true:
    println(MySingleton == evilTwin) // true

    // Do not compare data objects via ===.
    println(MySingleton === evilTwin) // false
}

fun createInstanceViaReflection(): MySingleton {
    // Kotlin reflection does not permit the instantiation of data objects.
    // This creates a new MySingleton instance "by force" (i.e. Java platform reflection)
    // Don't do this yourself!
    return (MySingleton.javaClass.declaredConstructors[0].apply { isAccessible = true } as Constructor<MySingleton>).newInstance()
}
```

The generated hashCode() function has behavior that is consistent with the equals() function, so that all runtime instances of a data object have the same hash code.

Differences between data objects and data classes

While data object and data class declarations are often used together and have some similarities, there are some functions that are not generated for a data object:

- No copy() function. Because a data object declaration is intended to be used as singleton objects, no copy() function is generated. The singleton pattern restricts the instantiation of a class to a single instance, which would be violated by allowing copies of the instance to be created.
- No componentN() function. Unlike a data class, a data object does not have any data properties. Since attempting to destructure such an object without data properties would not make sense, no componentN() functions are generated.

Using data objects with sealed hierarchies

data object declarations are a particularly useful for sealed hierarchies, like sealed classes or sealed interfaces, since they allow you to maintain symmetry with any data classes you may have defined alongside the object:

```
sealed interface ReadResult
```

```kotlin
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun printReadResult(r: ReadResult) {
    when(r) {
        is Number -> println("Num(${r.number}")
        is Text -> println("Txt(${r.text}")
        is EndOfFile -> println("EOF")
    }
}

fun main() {
    printReadResult(EndOfFile) // EOF
}
```

Companion objects

An object declaration inside a class can be marked with the companion keyword:

```kotlin
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

Members of the companion object can be called simply by using the class name as the qualifier:

```kotlin
val instance = MyClass.create()
```

The name of the companion object can be omitted, in which case the name Companion will be used:

```kotlin
class MyClass {
    companion object { }
}

val x = MyClass.Companion
```

Class members can access the private members of the corresponding companion object.

The name of a class used by itself (not as a qualifier to another name) acts as a reference to the companion object of the class (whether named or not):

```kotlin
class MyClass1 {
    companion object Named { }
}

val x = MyClass1

class MyClass2 {
    companion object { }
}

val y = MyClass2
```

Note that even though the members of companion objects look like static members in other languages, at runtime those are still instance members of real objects, and can, for example, implement interfaces:

```kotlin
interface Factory<T> {
    fun create(): T
}

class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}

val f: Factory<MyClass> = MyClass
```

However, on the JVM you can have members of companion objects generated as real static methods and fields if you use the @JvmStatic annotation. See the Java

interoperability section for more detail.

Semantic difference between object expressions and declarations

There is one important semantic difference between object expressions and object declarations:

- Object expressions are executed (and initialized) immediately, where they are used.
- Object declarations are initialized lazily, when accessed for the first time.
- A companion object is initialized when the corresponding class is loaded (resolved) that matches the semantics of a Java static initializer.

Delegation

The Delegation pattern has proven to be a good alternative to implementation inheritance, and Kotlin supports it natively requiring zero boilerplate code.

A class Derived can implement an interface Base by delegating all of its public members to a specified object:

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

The by-clause in the supertype list for Derived indicates that b will be stored internally in objects of Derived and the compiler will generate all the methods of Base that forward to b.

Overriding a member of an interface implemented by delegation

Overrides work as you expect: the compiler will use your override implementations instead of those in the delegate object. If you want to add override fun printMessage() { print("abc") } to Derived, the program would print abc instead of 10 when printMessage is called:

```
interface Base {
    fun printMessage()
    fun printMessageLine()
}

class BaseImpl(val x: Int) : Base {
    override fun printMessage() { print(x) }
    override fun printMessageLine() { println(x) }
}

class Derived(b: Base) : Base by b {
    override fun printMessage() { print("abc") }
}

fun main() {
    val b = BaseImpl(10)
    Derived(b).printMessage()
    Derived(b).printMessageLine()
}
```

Note, however, that members overridden in this way do not get called from the members of the delegate object, which can only access its own implementations of the interface members:

```
interface Base {
    val message: String
    fun print()
}
```

```kotlin
class BaseImpl(val x: Int) : Base {
    override val message = "BaseImpl: x = $x"
    override fun print() { println(message) }
}

class Derived(b: Base) : Base by b {
    // This property is not accessed from b's implementation of `print`
    override val message = "Message of Derived"
}

fun main() {
    val b = BaseImpl(10)
    val derived = Derived(b)
    derived.print()
    println(derived.message)
}
```

Learn more about delegated properties.

Delegated properties

With some common kinds of properties, even though you can implement them manually every time you need them, it is more helpful to implement them once, add them to a library, and reuse them later. For example:

- Lazy properties: the value is computed only on first access.
- Observable properties: listeners are notified about changes to this property.
- Storing properties in a map instead of a separate field for each property.

To cover these (and other) cases, Kotlin supports delegated properties:

```kotlin
class Example {
    var p: String by Delegate()
}
```

The syntax is: val/var <property name>: <Type> by <expression>. The expression after by is a delegate, because the get() (and set()) that correspond to the property will be delegated to its getValue() and setValue() methods. Property delegates don't have to implement an interface, but they have to provide a getValue() function (and setValue() for vars).

For example:

```kotlin
import kotlin.reflect.KProperty

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

When you read from p, which delegates to an instance of Delegate, the getValue() function from Delegate is called. Its first parameter is the object you read p from, and the second parameter holds a description of p itself (for example, you can take its name).

```kotlin
val e = Example()
println(e.p)
```

This prints:

Example@33a17727, thank you for delegating 'p' to me!

Similarly, when you assign to p, the setValue() function is called. The first two parameters are the same, and the third holds the value being assigned:

```kotlin
e.p = "NEW"
```

This prints:

NEW has been assigned to 'p' in Example@33a17727.

The specification of the requirements to the delegated object can be found below.

You can declare a delegated property inside a function or code block; it doesn't have to be a member of a class. Below you can find an example.

Standard delegates

The Kotlin standard library provides factory methods for several useful kinds of delegates.

Lazy properties

lazy() is a function that takes a lambda and returns an instance of Lazy<T>, which can serve as a delegate for implementing a lazy property. The first call to get() executes the lambda passed to lazy() and remembers the result. Subsequent calls to get() simply return the remembered result.

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

By default, the evaluation of lazy properties is synchronized: the value is computed only in one thread, but all threads will see the same value. If the synchronization of the initialization delegate is not required to allow multiple threads to execute it simultaneously, pass LazyThreadSafetyMode.PUBLICATION as a parameter to lazy().

If you're sure that the initialization will always happen in the same thread as the one where you use the property, you can use LazyThreadSafetyMode.NONE. It doesn't incur any thread-safety guarantees and related overhead.

Observable properties

Delegates.observable() takes two arguments: the initial value and a handler for modifications.

The handler is called every time you assign to the property (after the assignment has been performed). It has three parameters: the property being assigned to, the old value, and the new value:

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main() {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

If you want to intercept assignments and veto them, use vetoable() instead of observable(). The handler passed to vetoable will be called before the assignment of a new property value.

Delegating to another property

A property can delegate its getter and setter to another property. Such delegation is available for both top-level and class properties (member and extension). The delegate property can be:

- A top-level property

- A member or an extension property of the same class
- A member or an extension property of another class

To delegate a property to another property, use the :: qualifier in the delegate name, for example, this::delegate or MyClass::delegate.

```
var topLevelInt: Int = 0
class ClassWithDelegate(val anotherClassInt: Int)

class MyClass(var memberInt: Int, val anotherClassInstance: ClassWithDelegate) {
    var delegatedToMember: Int by this::memberInt
    var delegatedToTopLevel: Int by ::topLevelInt

    val delegatedToAnotherClass: Int by anotherClassInstance::anotherClassInt
}
var MyClass.extDelegated: Int by ::topLevelInt
```

This may be useful, for example, when you want to rename a property in a backward-compatible way: introduce a new property, annotate the old one with the @Deprecated annotation, and delegate its implementation.

```
class MyClass {
   var newName: Int = 0
   @Deprecated("Use 'newName' instead", ReplaceWith("newName"))
   var oldName: Int by this::newName
}
fun main() {
   val myClass = MyClass()
   // Notification: 'oldName: Int' is deprecated.
   // Use 'newName' instead
   myClass.oldName = 42
   println(myClass.newName) // 42
}
```

Storing properties in a map

One common use case is storing the values of properties in a map. This comes up often in applications for things like parsing JSON or performing other dynamic tasks. In this case, you can use the map instance itself as the delegate for a delegated property.

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

In this example, the constructor takes a map:

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

Delegated properties take values from this map through string keys, which are associated with the names of properties:

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}

fun main() {
    val user = User(mapOf(
        "name" to "John Doe",
        "age"  to 25
    ))
//sampleStart
    println(user.name) // Prints "John Doe"
    println(user.age)  // Prints 25
//sampleEnd
}
```

This also works for var's properties if you use a MutableMap instead of a read-only Map:

```kotlin
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

Local delegated properties

You can declare local variables as delegated properties. For example, you can make a local variable lazy:

```kotlin
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

The memoizedFoo variable will be computed on first access only. If someCondition fails, the variable won't be computed at all.

Property delegate requirements

For a read-only property (val), a delegate should provide an operator function getValue() with the following parameters:

- thisRef must be the same type as, or a supertype of, the property owner (for extension properties, it should be the type being extended).
- property must be of type KProperty<*> or its supertype.

getValue() must return the same type as the property (or its subtype).

```kotlin
class Resource

class Owner {
    val valResource: Resource by ResourceDelegate()
}

class ResourceDelegate {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return Resource()
    }
}
```

For a mutable property (var), a delegate has to additionally provide an operator function setValue() with the following parameters:

- thisRef must be the same type as, or a supertype of, the property owner (for extension properties, it should be the type being extended).
- property must be of type KProperty<*> or its supertype.
- value must be of the same type as the property (or its supertype).

```kotlin
class Resource

class Owner {
    var varResource: Resource by ResourceDelegate()
}

class ResourceDelegate(private var resource: Resource = Resource()) {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return resource
    }
    operator fun setValue(thisRef: Owner, property: KProperty<*>, value: Any?) {
        if (value is Resource) {
            resource = value
        }
    }
}
```

getValue() and/or setValue() functions can be provided either as member functions of the delegate class or as extension functions. The latter is handy when you need to delegate a property to an object that doesn't originally provide these functions. Both of the functions need to be marked with the operator keyword.

You can create delegates as anonymous objects without creating new classes, by using the interfaces ReadOnlyProperty and ReadWriteProperty from the Kotlin standard library. They provide the required methods: getValue() is declared in ReadOnlyProperty; ReadWriteProperty extends it and adds setValue(). This means you can pass a ReadWriteProperty whenever a ReadOnlyProperty is expected.

```
fun resourceDelegate(resource: Resource = Resource()): ReadWriteProperty<Any?, Resource> =
    object : ReadWriteProperty<Any?, Resource> {
        var curValue = resource
        override fun getValue(thisRef: Any?, property: KProperty<*>): Resource = curValue
        override fun setValue(thisRef: Any?, property: KProperty<*>, value: Resource) {
            curValue = value
        }
    }

val readOnlyResource: Resource by resourceDelegate()  // ReadWriteProperty as val
var readWriteResource: Resource by resourceDelegate()
```

Translation rules for delegated properties

Under the hood, the Kotlin compiler generates auxiliary properties for some kinds of delegated properties and then delegates to them.

> For the optimization purposes, the compiler does not generate auxiliary properties in several cases. Learn about the optimization on the example of delegating to another property.

For example, for the property prop it generates the hidden property prop$delegate, and the code of the accessors simply delegates to this additional property:

```
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler instead:
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

The Kotlin compiler provides all the necessary information about prop in the arguments: the first argument this refers to an instance of the outer class C, and this::prop is a reflection object of the KProperty type describing prop itself.

Optimized cases for delegated properties

The $delegate field will be omitted if a delegate is:

- A referenced property:

```
class C<Type> {
    private var impl: Type = ...
    var prop: Type by ::impl
}
```

- A named object:

```
object NamedObject {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String = ...
}

val s: String by NamedObject
```

- A final val property with a backing field and a default getter in the same module:

```
val impl: ReadOnlyProperty<Any?, String> = ...

class A {
    val s: String by impl
```

```
}
```

- A constant expression, enum entry, this, null. The example of this:

```
class A {
    operator fun getValue(thisRef: Any?, property: KProperty<*>) ...

    val s by this
}
```

Translation rules when delegating to another property

When delegating to another property, the Kotlin compiler generates immediate access to the referenced property. This means that the compiler doesn't generate the field prop$delegate. This optimization helps save memory.

Take the following code, for example:

```
class C<Type> {
    private var impl: Type = ...
    var prop: Type by ::impl
}
```

Property accessors of the prop variable invoke the impl variable directly, skipping the delegated property's getValueand setValue operators, and thus the KProperty reference object is not needed.

For the code above, the compiler generates the following code:

```
class C<Type> {
    private var impl: Type = ...

    var prop: Type
        get() = impl
        set(value) {
            impl = value
        }

    fun getProp$delegate(): Type = impl // This method is needed only for reflection
}
```

Providing a delegate

By defining the provideDelegate operator, you can extend the logic for creating the object to which the property implementation is delegated. If the object used on the right-hand side of by defines provideDelegate as a member or extension function, that function will be called to create the property delegate instance.

One of the possible use cases of provideDelegate is to check the consistency of the property upon its initialization.

For example, to check the property name before binding, you can write something like this:

```
class ResourceDelegate<T> : ReadOnlyProperty<MyUI, T> {
    override fun getValue(thisRef: MyUI, property: KProperty<*>): T { ... }
}

class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // create delegate
        return ResourceDelegate()
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

class MyUI {
    fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

    val image by bindResource(ResourceID.image_id)
```

```kotlin
    val text by bindResource(ResourceID.text_id)
}
```

The parameters of provideDelegate are the same as those of getValue:

- thisRef must be the same type as, or a supertype of, the property owner (for extension properties, it should be the type being extended);
- property must be of type KProperty<*> or its supertype.

The provideDelegate method is called for each property during the creation of the MyUI instance, and it performs the necessary validation right away.

Without this ability to intercept the binding between the property and its delegate, to achieve the same functionality you'd have to pass the property name explicitly, which isn't very convenient:

```kotlin
// Checking the property name without "provideDelegate" functionality
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
    checkProperty(this, propertyName)
    // create delegate
}
```

In the generated code, the provideDelegate method is called to initialize the auxiliary prop$delegate property. Compare the generated code for the property declaration val prop: Type by MyDelegate() with the generated code above (when the provideDelegate method is not present):

```kotlin
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler
// when the 'provideDelegate' function is available:
class C {
    // calling "provideDelegate" to create the additional "delegate" property
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

Note that the provideDelegate method affects only the creation of the auxiliary property and doesn't affect the code generated for the getter or the setter.

With the PropertyDelegateProvider interface from the standard library, you can create delegate providers without creating new classes.

```kotlin
val provider = PropertyDelegateProvider { thisRef: Any?, property ->
    ReadOnlyProperty<Any?, Int> {_, property -> 42 }
}
val delegate: Int by provider
```

Type aliases

Type aliases provide alternative names for existing types. If the type name is too long you can introduce a different shorter name and use the new one instead.

It's useful to shorten long generic types. For instance, it's often tempting to shrink collection types:

```kotlin
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

You can provide different aliases for function types:

```kotlin
typealias MyHandler = (Int, String, Any) -> Unit
```

```
typealias Predicate<T> = (T) -> Boolean
```

You can have new names for inner and nested classes:

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
typealias BInner = B.Inner
```

Type aliases do not introduce new types. They are equivalent to the corresponding underlying types. When you add typealias Predicate<T> and use Predicate<Int> in your code, the Kotlin compiler always expands it to (Int) -> Boolean. Thus you can pass a variable of your type whenever a general function type is required and vice versa:

```
typealias Predicate<T> = (T) -> Boolean

fun foo(p: Predicate<Int>) = p(42)

fun main() {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // prints "true"

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // prints "[1]"
}
```

Functions

Kotlin functions are declared using the fun keyword:

```
fun double(x: Int): Int {
    return 2 * x
}
```

Function usage

Functions are called using the standard approach:

```
val result = double(2)
```

Calling member functions uses dot notation:

```
Stream().read() // create instance of class Stream and call read()
```

Parameters

Function parameters are defined using Pascal notation - name: type. Parameters are separated using commas, and each parameter must be explicitly typed:

```
fun powerOf(number: Int, exponent: Int): Int { /*...*/ }
```

You can use a trailing comma when you declare function parameters:

```
fun powerOf(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }
```

Default arguments

Function parameters can have default values, which are used when you skip the corresponding argument. This reduces the number of overloads:

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

A default value is set by appending = to the type.

Overriding methods always use the base method's default parameter values. When overriding a method that has default parameter values, the default parameter values must be omitted from the signature:

```
open class A {
    open fun foo(i: Int = 10) { /*...*/ }
}

class B : A() {
    override fun foo(i: Int) { /*...*/ }  // No default value is allowed.
}
```

If a default parameter precedes a parameter with no default value, the default value can only be used by calling the function with named arguments:

```
fun foo(
    bar: Int = 0,
    baz: Int,
) { /*...*/ }

foo(baz = 1) // The default value bar = 0 is used
```

If the last argument after default parameters is a lambda, you can pass it either as a named argument or outside the parentheses:

```
fun foo(
    bar: Int = 0,
    baz: Int = 1,
    qux: () -> Unit,
) { /*...*/ }

foo(1) { println("hello") }     // Uses the default value baz = 1
foo(qux = { println("hello") }) // Uses both default values bar = 0 and baz = 1
foo { println("hello") }        // Uses both default values bar = 0 and baz = 1
```

Named arguments

You can name one or more of a function's arguments when calling it. This can be helpful when a function has many arguments and it's difficult to associate a value with an argument, especially if it's a boolean or null value.

When you use named arguments in a function call, you can freely change the order that they are listed in. If you want to use their default values, you can just leave these arguments out altogether.

Consider the reformat() function, which has 4 arguments with default values.

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ',
) { /*...*/ }
```

When calling this function, you don't have to name all its arguments:

```
reformat(
    "String!",
    false,
    upperCaseFirstLetter = false,
    divideByCamelHumps = true,
    '_'
```

```
)
```

You can skip all the ones with default values:

```
reformat("This is a long String!")
```

You are also able to skip specific arguments with default values, rather than omitting them all. However, after the first skipped argument, you must name all subsequent arguments:

```
reformat("This is a short String!", upperCaseFirstLetter = false, wordSeparator = '_')
```

You can pass a variable number of arguments (vararg) with names using the spread operator:

```
fun foo(vararg strings: String) { /*...*/ }

foo(strings = *arrayOf("a", "b", "c"))
```

When calling Java functions on the JVM, you can't use the named argument syntax because Java bytecode does not always preserve the names of function parameters.

Unit-returning functions

If a function does not return a useful value, its return type is Unit. Unit is a type with only one value - Unit. This value does not have to be returned explicitly:

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

The Unit return type declaration is also optional. The above code is equivalent to:

```
fun printHello(name: String?) { ... }
```

Single-expression functions

When a function returns a single expression, the curly braces can be omitted and the body is specified after a = symbol:

```
fun double(x: Int): Int = x * 2
```

Explicitly declaring the return type is optional when this can be inferred by the compiler:

```
fun double(x: Int) = x * 2
```

Explicit return types

Functions with block body must always specify return types explicitly, unless it's intended for them to return Unit, in which case specifying the return type is optional.

Kotlin does not infer return types for functions with block bodies because such functions may have complex control flow in the body, and the return type will be non-obvious to the reader (and sometimes even for the compiler).

Variable number of arguments (varargs)

You can mark a parameter of a function (usually the last one) with the vararg modifier:

```
fun <T> asList(vararg ts: T): List<T> {
```

```
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

In this case, you can pass a variable number of arguments to the function:

```
val list = asList(1, 2, 3)
```

Inside a function, a vararg-parameter of type T is visible as an array of T, as in the example above, where the ts variable has type Array<out T>.

Only one parameter can be marked as vararg. If a vararg parameter is not the last one in the list, values for the subsequent parameters can be passed using named argument syntax, or, if the parameter has a function type, by passing a lambda outside the parentheses.

When you call a vararg-function, you can pass arguments individually, for example asList(1, 2, 3). If you already have an array and want to pass its contents to the function, use the spread operator (prefix the array with *):

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

If you want to pass a primitive type array into vararg, you need to convert it to a regular (typed) array using the toTypedArray() function:

```
val a = intArrayOf(1, 2, 3) // IntArray is a primitive type array
val list = asList(-1, 0, *a.toTypedArray(), 4)
```

Infix notation

Functions marked with the infix keyword can also be called using the infix notation (omitting the dot and the parentheses for the call). Infix functions must meet the following requirements:

- They must be member functions or extension functions.
- They must have a single parameter.
- The parameter must not accept variable number of arguments and must have no default value.

```
infix fun Int.shl(x: Int): Int { ... }

// calling the function using the infix notation
1 shl 2

// is the same as
1.shl(2)
```

Infix function calls have lower precedence than arithmetic operators, type casts, and the rangeTo operator. The following expressions are equivalent:

- 1 shl 2 + 3 is equivalent to 1 shl (2 + 3)
- 0 until n * 2 is equivalent to 0 until (n * 2)
- xs union ys as Set<*> is equivalent to xs union (ys as Set<*>)

On the other hand, an infix function call's precedence is higher than that of the boolean operators && and ||, is- and in-checks, and some other operators. These expressions are equivalent as well:

- a && b xor c is equivalent to a && (b xor c)
- a xor b in c is equivalent to (a xor b) in c

Note that infix functions always require both the receiver and the parameter to be specified. When you're calling a method on the current receiver using the infix notation, use this explicitly. This is required to ensure unambiguous parsing.

```
class MyStringCollection {
```

```
    infix fun add(s: String) { /*...*/ }

    fun build() {
        this add "abc"   // Correct
        add("abc")       // Correct
        //add "abc"        // Incorrect: the receiver must be specified
    }
}
```

Function scope

Kotlin functions can be declared at the top level in a file, meaning you do not need to create a class to hold a function, which you are required to do in languages such as Java, C#, and Scala (top level definition is available since Scala 3). In addition to top level functions, Kotlin functions can also be declared locally as member functions and extension functions.

Local functions

Kotlin supports local functions, which are functions inside other functions:

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}
```

A local function can access local variables of outer functions (the closure). In the case above, visited can be a local variable:

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

Member functions

A member function is a function that is defined inside a class or object:

```
class Sample {
    fun foo() { print("Foo") }
}
```

Member functions are called with dot notation:

```
Sample().foo() // creates instance of class Sample and calls foo
```

For more information on classes and overriding members see Classes and Inheritance.

Generic functions

Functions can have generic parameters, which are specified using angle brackets before the function name:

```
fun <T> singletonList(item: T): List<T> { /*...*/ }
```

For more information on generic functions, see Generics.

Tail recursive functions

Kotlin supports a style of functional programming known as tail recursion. For some algorithms that would normally use loops, you can use a recursive function instead without the risk of stack overflow. When a function is marked with the tailrec modifier and meets the required formal conditions, the compiler optimizes out the recursion, leaving behind a fast and efficient loop based version instead:

```
val eps = 1E-10 // "good enough", could be 10^-15

tailrec fun findFixPoint(x: Double = 1.0): Double =
    if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

This code calculates the fixpoint of cosine, which is a mathematical constant. It simply calls Math.cos repeatedly starting at 1.0 until the result no longer changes, yielding a result of 0.7390851332151611 for the specified eps precision. The resulting code is equivalent to this more traditional style:

```
val eps = 1E-10 // "good enough", could be 10^-15

private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (Math.abs(x - y) < eps) return x
        x = Math.cos(x)
    }
}
```

To be eligible for the tailrec modifier, a function must call itself as the last operation it performs. You cannot use tail recursion when there is more code after the recursive call, within try/catch/finally blocks, or on open functions. Currently, tail recursion is supported by Kotlin for the JVM and Kotlin/Native.

See also:

- Inline functions
- Extension functions
- Higher-order functions and lambdas

High-order functions and lambdas

Kotlin functions are first-class, which means they can be stored in variables and data structures, and can be passed as arguments to and returned from other higher-order functions. You can perform any operations on functions that are possible for other non-function values.

To facilitate this, Kotlin, as a statically typed programming language, uses a family of function types to represent functions, and provides a set of specialized language constructs, such as lambda expressions.

Higher-order functions

A higher-order function is a function that takes functions as parameters, or returns a function.

A good example of a higher-order function is the functional programming idiom fold for collections. It takes an initial accumulator value and a combining function and builds its return value by consecutively combining the current accumulator value with each collection element, replacing the accumulator value each time:

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

In the code above, the combine parameter has the function type (R, T) -> R, so it accepts a function that takes two arguments of types R and T and returns a value of type R. It is invoked inside the for loop, and the return value is then assigned to accumulator.

To call fold, you need to pass an instance of the function type to it as an argument, and lambda expressions (described in more detail below) are widely used for this purpose at higher-order function call sites:

```
fun main() {
    //sampleStart
    val items = listOf(1, 2, 3, 4, 5)

    // Lambdas are code blocks enclosed in curly braces.
    items.fold(0, {
        // When a lambda has parameters, they go first, followed by '->'
        acc: Int, i: Int ->
        print("acc = $acc, i = $i, ")
        val result = acc + i
        println("result = $result")
        // The last expression in a lambda is considered the return value:
        result
    })

    // Parameter types in a lambda are optional if they can be inferred:
    val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })

    // Function references can also be used for higher-order function calls:
    val product = items.fold(1, Int::times)
    //sampleEnd
    println("joinedToString = $joinedToString")
    println("product = $product")
}
```

Function types

Kotlin uses function types, such as (Int) -> String, for declarations that deal with functions: val onClick: () -> Unit = ...

These types have a special notation that corresponds to the signatures of the functions - their parameters and return values:

- All function types have a parenthesized list of parameter types and a return type: (A, B) -> C denotes a type that represents functions that take two arguments of types A and B and return a value of type C. The list of parameter types may be empty, as in () -> A. The Unit return type cannot be omitted.
- Function types can optionally have an additional receiver type, which is specified before the dot in the notation: the type A.(B) -> C represents functions that can be called on a receiver object A with a parameter B and return a value C. Function literals with receiver are often used along with these types.
- Suspending functions belong to a special kind of function type that have a suspend modifier in their notation, such as suspend () -> Unit or suspend A.(B) -> C.

The function type notation can optionally include names for the function parameters: (x: Int, y: Int) -> Point. These names can be used for documenting the meaning of the parameters.

To specify that a function type is nullable, use parentheses as follows: ((Int, Int) -> Int)?.

Function types can also be combined using parentheses: (Int) -> ((Int) -> Unit).

> The arrow notation is right-associative, (Int) -> (Int) -> Unit is equivalent to the previous example, but not to ((Int) -> (Int)) -> Unit.

You can also give a function type an alternative name by using a type alias:

```
typealias ClickHandler = (Button, ClickEvent) -> Unit
```

Instantiating a function type

There are several ways to obtain an instance of a function type:

- Use a code block within a function literal, in one of the following forms:
  - a lambda expression: { a, b -> a + b },
  - an anonymous function: fun(s: String): Int { return s.toIntOrNull() ?: 0 }

  Function literals with receiver can be used as values of function types with receiver.

- Use a callable reference to an existing declaration:
  - a top-level, local, member, or extension function: ::isOdd, String::toInt,
  - a top-level, member, or extension property: List<Int>::size,
  - a constructor: ::Regex

  These include bound callable references that point to a member of a particular instance: foo::toString.

- Use instances of a custom class that implements a function type as an interface:

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

The compiler can infer the function types for variables if there is enough information:

```
val a = { i: Int -> i + 1 } // The inferred type is (Int) -> Int
```

Non-literal values of function types with and without a receiver are interchangeable, so the receiver can stand in for the first parameter, and vice versa. For instance, a value of type (A, B) -> C can be passed or assigned where a value of type A.(B) -> C is expected, and the other way around:

```
fun main() {
    //sampleStart
    val repeatFun: String.(Int) -> String = { times -> this.repeat(times) }
    val twoParameters: (String, Int) -> String = repeatFun // OK

    fun runTransformation(f: (String, Int) -> String): String {
        return f("hello", 3)
    }
    val result = runTransformation(repeatFun) // OK
    //sampleEnd
    println("result = $result")
}
```

A function type with no receiver is inferred by default, even if a variable is initialized with a reference to an extension function. To alter that, specify the variable type explicitly.

Invoking a function type instance

A value of a function type can be invoked by using its invoke(...) operator: f.invoke(x) or just f(x).

If the value has a receiver type, the receiver object should be passed as the first argument. Another way to invoke a value of a function type with receiver is to prepend it with the receiver object, as if the value were an extension function: 1.foo(2).

Example:

```
fun main() {
    //sampleStart
    val stringPlus: (String, String) -> String = String::plus
    val intPlus: Int.(Int) -> Int = Int::plus

    println(stringPlus.invoke("<-", "->"))
    println(stringPlus("Hello, ", "world!"))

    println(intPlus.invoke(1, 1))
    println(intPlus(1, 2))
    println(2.intPlus(3)) // extension-like call
    //sampleEnd
}
```

Inline functions

Sometimes it is beneficial to use inline functions, which provide flexible control flow, for higher-order functions.

Lambda expressions and anonymous functions

Lambda expressions and anonymous functions are function literals. Function literals are functions that are not declared but are passed immediately as an expression. Consider the following example:

```
max(strings, { a, b -> a.length < b.length })
```

The function max is a higher-order function, as it takes a function value as its second argument. This second argument is an expression that is itself a function, called a function literal, which is equivalent to the following named function:

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Lambda expression syntax

The full syntactic form of lambda expressions is as follows:

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

- A lambda expression is always surrounded by curly braces.
- Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations.
- The body goes after the ->.
- If the inferred return type of the lambda is not Unit, the last (or possibly single) expression inside the lambda body is treated as the return value.

If you leave all the optional annotations out, what's left looks like this:

```
val sum = { x: Int, y: Int -> x + y }
```

Passing trailing lambdas

According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses:

```
val product = items.fold(1) { acc, e -> acc * e }
```

Such syntax is also known as trailing lambda.

If the lambda is the only argument in that call, the parentheses can be omitted entirely:

```
run { println("...") }
```

it: implicit name of a single parameter

It's very common for a lambda expression to have only one parameter.

If the compiler can parse the signature without any parameters, the parameter does not need to be declared and -> can be omitted. The parameter will be implicitly declared under the name it:

```
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

Returning a value from a lambda expression

You can explicitly return a value from the lambda using the qualified return syntax. Otherwise, the value of the last expression is implicitly returned.

Therefore, the two following snippets are equivalent:

```
ints.filter {
    val shouldFilter = it > 0
```

```
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

This convention, along with passing a lambda expression outside of parentheses, allows for LINQ-style code:

```
strings.filter { it.length == 5 }.sortedBy { it }.map { it.uppercase() }
```

Underscore for unused variables

If the lambda parameter is unused, you can place an underscore instead of its name:

```
map.forEach { (_, value) -> println("$value!") }
```

Destructuring in lambdas

Destructuring in lambdas is described as a part of destructuring declarations.

Anonymous functions

The lambda expression syntax above is missing one thing – the ability to specify the function's return type. In most cases, this is unnecessary because the return type can be inferred automatically. However, if you do need to specify it explicitly, you can use an alternative syntax: an anonymous function.

```
fun(x: Int, y: Int): Int = x + y
```

An anonymous function looks very much like a regular function declaration, except its name is omitted. Its body can be either an expression (as shown above) or a block:

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

The parameters and the return type are specified in the same way as for regular functions, except the parameter types can be omitted if they can be inferred from the context:

```
ints.filter(fun(item) = item > 0)
```

The return type inference for anonymous functions works just like for normal functions: the return type is inferred automatically for anonymous functions with an expression body, but it has to be specified explicitly (or is assumed to be Unit) for anonymous functions with a block body.

> When passing anonymous functions as parameters, place them inside the parentheses. The shorthand syntax that allows you to leave the function outside the parentheses works only for lambda expressions.

Another difference between lambda expressions and anonymous functions is the behavior of non-local returns. A return statement without a label always returns from the function declared with the fun keyword. This means that a return inside a lambda expression will return from the enclosing function, whereas a return inside an anonymous function will return from the anonymous function itself.

Closures

A lambda expression or anonymous function (as well as a local function and an object expression) can access its closure, which includes the variables declared in the outer scope. The variables captured in the closure can be modified in the lambda:

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
```

```
print(sum)
```

Function literals with receiver

Function types with receiver, such as A.(B) -> C, can be instantiated with a special form of function literals – function literals with receiver.

As mentioned above, Kotlin provides the ability to call an instance of a function type with receiver while providing the receiver object.

Inside the body of the function literal, the receiver object passed to a call becomes an implicit this, so that you can access the members of that receiver object without any additional qualifiers, or access the receiver object using a this expression.

This behavior is similar to that of extension functions, which also allow you to access the members of the receiver object inside the function body.

Here is an example of a function literal with receiver along with its type, where plus is called on the receiver object:

```
val sum: Int.(Int) -> Int = { other -> plus(other) }
```

The anonymous function syntax allows you to specify the receiver type of a function literal directly. This can be useful if you need to declare a variable of a function type with receiver, and then to use it later.

```
val sum = fun Int.(other: Int): Int = this + other
```

Lambda expressions can be used as function literals with receiver when the receiver type can be inferred from the context. One of the most important examples of their usage is type-safe builders:

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()  // create the receiver object
    html.init()        // pass the receiver object to the lambda
    return html
}

html {       // lambda with receiver begins here
    body()   // calling a method on the receiver object
}
```

Inline functions

Using higher-order functions imposes certain runtime penalties: each function is an object, and it captures a closure. A closure is a scope of variables that can be accessed in the body of the function. Memory allocations (both for function objects and classes) and virtual calls introduce runtime overhead.

But it appears that in many cases this kind of overhead can be eliminated by inlining the lambda expressions. The functions shown below are good examples of this situation. The lock() function could be easily inlined at call-sites. Consider the following case:

```
lock(l) { foo() }
```

Instead of creating a function object for the parameter and generating a call, the compiler could emit the following code:

```
l.lock()
try {
    foo()
} finally {
    l.unlock()
}
```

To make the compiler do this, mark the lock() function with the inline modifier:

```
inline fun <T> lock(lock: Lock, body: () -> T): T { ... }
```

The inline modifier affects both the function itself and the lambdas passed to it: all of those will be inlined into the call site.

Inlining may cause the generated code to grow. However, if you do it in a reasonable way (avoiding inlining large functions), it will pay off in performance, especially at "megamorphic" call-sites inside loops.

noinline

If you don't want all of the lambdas passed to an inline function to be inlined, mark some of your function parameters with the noinline modifier:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) { ... }
```

Inlinable lambdas can only be called inside inline functions or passed as inlinable arguments. noinline lambdas, however, can be manipulated in any way you like, including being stored in fields or passed around.

> If an inline function has no inlinable function parameters and no reified type parameters, the compiler will issue a warning, since inlining such functions is very unlikely to be beneficial (you can use the @Suppress("NOTHING_TO_INLINE") annotation to suppress the warning if you are sure the inlining is needed).

Non-local returns

In Kotlin, you can only use a normal, unqualified return to exit a named function or an anonymous function. To exit a lambda, use a label. A bare return is forbidden inside a lambda because a lambda cannot make the enclosing function return:

```
fun ordinaryFunction(block: () -> Unit) {
    println("hi!")
}
//sampleStart
fun foo() {
    ordinaryFunction {
        return // ERROR: cannot make `foo` return here
    }
}
//sampleEnd
fun main() {
    foo()
}
```

But if the function the lambda is passed to is inlined, the return can be inlined, as well. So it is allowed:

```
inline fun inlined(block: () -> Unit) {
    println("hi!")
}
//sampleStart
fun foo() {
    inlined {
        return // OK: the lambda is inlined
    }
}
//sampleEnd
fun main() {
    foo()
}
```

Such returns (located in a lambda, but exiting the enclosing function) are called non-local returns. This sort of construct usually occurs in loops, which inline functions often enclose:

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // returns from hasZeros
    }
    return false
}
```

Note that some inline functions may call the lambdas passed to them as parameters not directly from the function body, but from another execution context, such as a local object or a nested function. In such cases, non-local control flow is also not allowed in the lambdas. To indicate that the lambda parameter of the inline function cannot use non-local returns, mark the lambda parameter with the crossinline modifier:

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

break and continue are not yet available in inlined lambdas, but we are planning to support them, too.

Reified type parameters

Sometimes you need to access a type passed as a parameter:

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

Here, you walk up a tree and use reflection to check whether a node has a certain type. It's all fine, but the call site is not very pretty:

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

A better solution would be to simply pass a type to this function. You can call it as follows:

```
treeNode.findParentOfType<MyTreeNode>()
```

To enable this, inline functions support reified type parameters, so you can write something like this:

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

The code above qualifies the type parameter with the reified modifier to make it accessible inside the function, almost as if it were a normal class. Since the function is inlined, no reflection is needed and normal operators like !is and as are now available for you to use. Also, you can call the function as shown above: myTree.findParentOfType<MyTreeNodeType>().

Though reflection may not be needed in many cases, you can still use it with a reified type parameter:

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

Normal functions (not marked as inline) cannot have reified parameters. A type that does not have a run-time representation (for example, a non-reified type parameter or a fictitious type like Nothing) cannot be used as an argument for a reified type parameter.

Inline properties

The inline modifier can be used on accessors of properties that don't have backing fields. You can annotate individual property accessors:

```
val foo: Foo
    inline get() = Foo()
```

```kotlin
var bar: Bar
    get() = ...
    inline set(v) { ... }
```

You can also annotate an entire property, which marks both of its accessors as inline:

```kotlin
inline var bar: Bar
    get() = ...
    set(v) { ... }
```

At the call site, inline accessors are inlined as regular inline functions.

Restrictions for public API inline functions

When an inline function is public or protected but is not a part of a private or internal declaration, it is considered a module's public API. It can be called in other modules and is inlined at such call sites as well.

This imposes certain risks of binary incompatibility caused by changes in the module that declares an inline function in case the calling module is not re-compiled after the change.

To eliminate the risk of such incompatibility being introduced by a change in a non-public API of a module, public API inline functions are not allowed to use non-public-API declarations, i.e. private and internal declarations and their parts, in their bodies.

An internal declaration can be annotated with @PublishedApi, which allows its use in public API inline functions. When an internal inline function is marked as @PublishedApi, its body is checked too, as if it were public.

Operator overloading

Kotlin allows you to provide custom implementations for the predefined set of operators on types. These operators have predefined symbolic representation (like + or *) and precedence. To implement an operator, provide a member function or an extension function with a specific name for the corresponding type. This type becomes the left-hand side type for binary operations and the argument type for the unary ones.

To overload an operator, mark the corresponding function with the operator modifier:

```kotlin
interface IndexedContainer {
    operator fun get(index: Int)
}
```

When overriding your operator overloads, you can omit operator:

```kotlin
class OrdersList: IndexedContainer {
    override fun get(index: Int) { /*...*/ }
}
```

Unary operations

Unary prefix operators

Expression	Translated to
+a	a.unaryPlus()
-a	a.unaryMinus()

Expression	Translated to
!a	a.not()

This table says that when the compiler processes, for example, an expression +a, it performs the following steps:

- Determines the type of a, let it be T.
- Looks up a function unaryPlus() with the operator modifier and no parameters for the receiver T, that means a member function or an extension function.
- If the function is absent or ambiguous, it is a compilation error.
- If the function is present and its return type is R, the expression +a has type R.

> These operations, as well as all the others, are optimized for basic types and do not introduce overhead of function calls for them.

As an example, here's how you can overload the unary minus operator:

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
   println(-point)  // prints "Point(x=-10, y=-20)"
}
```

Increments and decrements

Expression	Translated to
a++	a.inc() + see below
a--	a.dec() + see below

The inc() and dec() functions must return a value, which will be assigned to the variable on which the ++ or -- operation was used. They shouldn't mutate the object on which the inc or dec was invoked.

The compiler performs the following steps for resolution of an operator in the postfix form, for example a++:

- Determines the type of a, let it be T.
- Looks up a function inc() with the operator modifier and no parameters, applicable to the receiver of type T.
- Checks that the return type of the function is a subtype of T.

The effect of computing the expression is:

- Store the initial value of a to a temporary storage a0.
- Assign the result of a0.inc() to a.
- Return a0 as the result of the expression.

For a-- the steps are completely analogous.

For the prefix forms ++a and --a resolution works the same way, and the effect is:

- Assign the result of a.inc() to a.

* Return the new value of a as a result of the expression.

Binary operations

Arithmetic operators

Expression	Translated to
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

For the operations in this table, the compiler just resolves the expression in the Translated to column.

Below is an example Counter class that starts at a given value and can be incremented using the overloaded + operator:

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

in operator

Expression	Translated to
a in b	b.contains(a)
a !in b	!b.contains(a)

For in and !in the procedure is the same, but the order of arguments is reversed.

Indexed access operator

Expression	Translated to
a[i]	a.get(i)

Expression	Translated to
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

Square brackets are translated to calls to get and set with appropriate numbers of arguments.

invoke operator

Expression	Translated to
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

Parentheses are translated to calls to invoke with appropriate number of arguments.

Augmented assignments

Expression	Translated to
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)

For the assignment operations, for example a += b, the compiler performs the following steps:

- If the function from the right column is available:
  - If the corresponding binary function (that means plus() for plusAssign()) is available too, a is a mutable variable, and the return type of plus is a subtype of the type of a, report an error (ambiguity).
  - Make sure its return type is Unit, and report an error otherwise.
  - Generate code for a.plusAssign(b).
- Otherwise, try to generate code for a = a + b (this includes a type check: the type of a + b must be a subtype of a).

> Assignments are NOT expressions in Kotlin.

Equality and inequality operators

Expression	Translated to
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))

These operators only work with the function equals(other: Any?): Boolean, which can be overridden to provide custom equality check implementation. Any other function with the same name (like equals(other: Foo)) will not be called.

> === and !== (identity checks) are not overloadable, so no conventions exist for them.

The == operation is special: it is translated to a complex expression that screens for null's. null == null is always true, and x == null for a non-null x is always false and won't invoke x.equals().

Comparison operators

Expression	Translated to
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

All comparisons are translated into calls to compareTo, that is required to return Int.

Property delegation operators

provideDelegate, getValue and setValue operator functions are described in Delegated properties.

Infix calls for named functions

You can simulate custom infix operations by using infix function calls.

Type-safe builders

By using well-named functions as builders in combination with function literals with receiver it is possible to create type-safe, statically-typed builders in Kotlin.

Type-safe builders allow creating Kotlin-based domain-specific languages (DSLs) suitable for building complex hierarchical data structures in a semi-declarative way. Sample use cases for the builders are:

- Generating markup with Kotlin code, such as HTML or XML
- Configuring routes for a web server: Ktor

Consider the following code:

```
import com.example.html.* // see declarations below

fun result() =
    html {
        head {
            title {+"XML encoding with Kotlin"}
        }
        body {
            h1 {+"XML encoding with Kotlin"}
            p  {+"this format can be used as an alternative markup to XML"}

            // an element with attributes and text content
            a(href = "https://kotlinlang.org") {+"Kotlin"}

            // mixed content
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "https://kotlinlang.org") {+"Kotlin"}
                +"project"
            }
            p {+"some text"}

            // content generated by
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

This is completely legitimate Kotlin code. You can play with this code online (modify it and run in the browser) here.

How it works

Assume that you need to implement a type-safe builder in Kotlin. First of all, define the model you want to build. In this case you need to model HTML tags. It is easily done with a bunch of classes. For example, HTML is a class that describes the <html> tag defining children like <head> and <body>. (See its declaration below.)

Now, let's recall why you can say something like this in the code:

```
html {
 // ...
}
```

html is actually a function call that takes a lambda expression as an argument. This function is defined as follows:

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

This function takes one parameter named init, which is itself a function. The type of the function is HTML.() -> Unit, which is a function type with receiver. This means that you need to pass an instance of type HTML (a receiver) to the function, and you can call members of that instance inside the function.

The receiver can be accessed through the this keyword:

```
html {
    this.head { ... }
    this.body { ... }
}
```

(head and body are member functions of HTML.)

Now, this can be omitted, as usual, and you get something that looks very much like a builder already:

```
html {
    head { ... }
    body { ... }
}
```

So, what does this call do? Let's look at the body of html function as defined above. It creates a new instance of HTML, then it initializes it by calling the function that is passed as an argument (in this example this boils down to calling head and body on the HTML instance), and then it returns this instance. This is exactly what a builder should do.

The head and body functions in the HTML class are defined similarly to html. The only difference is that they add the built instances to the children collection of the enclosing HTML instance:

```
fun head(init: Head.() -> Unit): Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit): Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
}
```

Actually these two functions do just the same thing, so you can have a generic version, initTag:

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

So, now your functions are very simple:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

And you can use them to build <head> and <body> tags.

One other thing to be discussed here is how you add text to tag bodies. In the example above you say something like:

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}
```

So basically, you just put a string inside a tag body, but there is this little + in front of it, so it is a function call that invokes a prefix unaryPlus() operation. That operation is actually defined by an extension function unaryPlus() that is a member of the TagWithText abstract class (a parent of Title):

```
operator fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

So, what the prefix + does here is wrapping a string into an instance of TextElement and adding it to the children collection, so that it becomes a proper part of the tag tree.

All this is defined in a package com.example.html that is imported at the top of the builder example above. In the last section you can read through the full definition of this package.

Scope control: @DslMarker

When using DSLs, one might have come across the problem that too many functions can be called in the context. You can call methods of every available implicit receiver inside a lambda and therefore get an inconsistent result, like the tag head inside another head:

```
html {
    head {
        head {} // should be forbidden
    }
    // ...
}
```

In this example only members of the nearest implicit receiver this@head must be available; head() is a member of the outer receiver this@html, so it must be illegal to call it.

To address this problem, there is a special mechanism to control receiver scope.

To make the compiler start controlling scopes you only have to annotate the types of all receivers used in the DSL with the same marker annotation. For instance, for HTML Builders you declare an annotation @HTMLTagMarker:

```
@DslMarker
annotation class HtmlTagMarker
```

An annotation class is called a DSL marker if it is annotated with the @DslMarker annotation.

In our DSL all the tag classes extend the same superclass Tag. It's enough to annotate only the superclass with @HtmlTagMarker and after that the Kotlin compiler will treat all the inherited classes as annotated:

```
@HtmlTagMarker
abstract class Tag(val name: String) { ... }
```

You don't have to annotate the HTML or Head classes with @HtmlTagMarker because their superclass is already annotated:

class HTML() : Tag("html") { ... } class Head() : Tag("head") { ... }

After you've added this annotation, the Kotlin compiler knows which implicit receivers are part of the same DSL and allows to call members of the nearest receivers only:

```
html {
    head {
        head { } // error: a member of outer receiver
    }
    // ...
}
```

Note that it's still possible to call the members of the outer receiver, but to do that you have to specify this receiver explicitly:

```
html {
    head {
        this@html.head { } // possible
    }
    // ...
}
```

Full definition of the com.example.html package

This is how the package com.example.html is defined (only the elements used in the example above). It builds an HTML tree. It makes heavy use of extension

functions and lambdas with receiver.

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }

    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for ((attr, value) in attributes) {
            builder.append(" $attr=\"$value\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body : BodyTag("body")
class B : BodyTag("b")
```

```kotlin
class P : BodyTag("p")
class H1 : BodyTag("h1")

class A : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

Using builders with builder type inference

Kotlin supports builder type inference (or builder inference), which can come in useful when you are working with generic builders. It helps the compiler infer the type arguments of a builder call based on the type information about other calls inside its lambda argument.

Consider this example of buildMap() usage:

```kotlin
fun addEntryToMap(baseMap: Map<String, Number>, additionalEntry: Pair<String, Int>?) {
   val myMap = buildMap {
       putAll(baseMap)
       if (additionalEntry != null) {
           put(additionalEntry.first, additionalEntry.second)
       }
   }
}
```

There is not enough type information here to infer type arguments in a regular way, but builder inference can analyze the calls inside the lambda argument. Based on the type information about putAll() and put() calls, the compiler can automatically infer type arguments of the buildMap() call into String and Number. Builder inference allows to omit type arguments while using generic builders.

Writing your own builders

Requirements for enabling builder inference

> Before Kotlin 1.7.0, enabling builder inference for a builder function required -Xenable-builder-inference compiler option. In 1.7.0 the option is enabled by default.

To let builder inference work for your own builder, make sure its declaration has a builder lambda parameter of a function type with a receiver. There are also two requirements for the receiver type:

1. It should use the type arguments that builder inference is supposed to infer. For example:

   ```kotlin
   fun <V> buildList(builder: MutableList<V>.() -> Unit) { ... }
   ```

   > Note that passing the type parameter's type directly like fun <T> myBuilder(builder: T.() -> Unit) is not yet supported.

2. It should provide public members or extensions that contain the corresponding type parameters in their signature. For example:

   ```kotlin
   class ItemHolder<T> {
       private val items = mutableListOf<T>()

       fun addItem(x: T) {
           items.add(x)
   ```

```
    }

    fun getLastItem(): T? = items.lastOrNull()
}

fun <T> ItemHolder<T>.addAllItems(xs: List<T>) {
    xs.forEach { addItem(it) }
}

fun <T> itemHolderBuilder(builder: ItemHolder<T>.() -> Unit): ItemHolder<T> =
    ItemHolder<T>().apply(builder)

fun test(s: String) {
    val itemHolder1 = itemHolderBuilder { // Type of itemHolder1 is ItemHolder<String>
        addItem(s)
    }
    val itemHolder2 = itemHolderBuilder { // Type of itemHolder2 is ItemHolder<String>
        addAllItems(listOf(s))
    }
    val itemHolder3 = itemHolderBuilder { // Type of itemHolder3 is ItemHolder<String?>
        val lastItem: String? = getLastItem()
        // ...
    }
}
```

Supported features

Builder inference supports:

- Inferring several type arguments

```
fun <K, V> myBuilder(builder: MutableMap<K, V>.() -> Unit): Map<K, V> { ... }
```

- Inferring type arguments of several builder lambdas within one call including interdependent ones

```
fun <K, V> myBuilder(
    listBuilder: MutableList<V>.() -> Unit,
    mapBuilder: MutableMap<K, V>.() -> Unit
): Pair<List<V>, Map<K, V>> =
    mutableListOf<V>().apply(listBuilder) to mutableMapOf<K, V>().apply(mapBuilder)

fun main() {
    val result = myBuilder(
        { add(1) },
        { put("key", 2) }
    )
    // result has Pair<List<Int>, Map<String, Int>> type
}
```

- Inferring type arguments whose type parameters are lambda's parameter or return types

```
fun <K, V> myBuilder1(
    mapBuilder: MutableMap<K, V>.() -> K
): Map<K, V> = mutableMapOf<K, V>().apply { mapBuilder() }

fun <K, V> myBuilder2(
    mapBuilder: MutableMap<K, V>.(K) -> Unit
): Map<K, V> = mutableMapOf<K, V>().apply { mapBuilder(2 as K) }

fun main() {
    // result1 has the Map<Long, String> type inferred
    val result1 = myBuilder1 {
        put(1L, "value")
        2
    }
    val result2 = myBuilder2 {
        put(1, "value 1")
        // You can use `it` as "postponed type variable" type
        // See the details in the section below
        put(it, "value 2")
    }
}
```

How builder inference works

Postponed type variables

Builder inference works in terms of postponed type variables, which appear inside the builder lambda during builder inference analysis. A postponed type variable is a type argument's type, which is in the process of inferring. The compiler uses it to collect type information about the type argument.

Consider the example with buildList():

```
val result = buildList {
    val x = get(0)
}
```

Here x has a type of postponed type variable: the get() call returns a value of type E, but E itself is not yet fixed. At this moment, a concrete type for E is unknown.

When a value of a postponed type variable gets associated with a concrete type, builder inference collects this information to infer the resulting type of the corresponding type argument at the end of the builder inference analysis. For example:

```
val result = buildList {
    val x = get(0)
    val y: String = x
} // result has the List<String> type inferred
```

After the postponed type variable gets assigned to a variable of the String type, builder inference gets the information that x is a subtype of String. This assignment is the last statement in the builder lambda, so the builder inference analysis ends with the result of inferring the type argument E into String.

Note that you can always call equals(), hashCode(), and toString() functions with a postponed type variable as a receiver.

Contributing to builder inference results

Builder inference can collect different varieties of type information that contribute to the analysis result. It considers:

- Calling methods on a lambda's receiver that use the type parameter's type

```
val result = buildList {
    // Type argument is inferred into String based on the passed "value" argument
    add("value")
} // result has the List<String> type inferred
```

- Specifying the expected type for calls that return the type parameter's type

```
val result = buildList {
    // Type argument is inferred into Float based on the expected type
    val x: Float = get(0)
} // result has the List<Float> type
```

```
class Foo<T> {
    val items = mutableListOf<T>()
}

fun <K> myBuilder(builder: Foo<K>.() -> Unit): Foo<K> = Foo<K>().apply(builder)

fun main() {
    val result = myBuilder {
        val x: List<CharSequence> = items
        // ...
    } // result has the Foo<CharSequence> type
}
```

- Passing postponed type variables' types into methods that expect concrete types

```
fun takeMyLong(x: Long) { ... }

fun String.isMoreThat3() = length > 3

fun takeListOfStrings(x: List<String>) { ... }
```

```
fun main() {
    val result1 = buildList {
        val x = get(0)
        takeMyLong(x)
    } // result1 has the List<Long> type

    val result2 = buildList {
        val x = get(0)
        val isLong = x.isMoreThat3()
    // ...
    } // result2 has the List<String> type

    val result3 = buildList {
        takeListOfStrings(this)
    } // result3 has the List<String> type
}
```

- Taking a callable reference to the lambda receiver's member

```
fun main() {
    val result = buildList {
        val x: KFunction1<Int, Float> = ::get
    } // result has the List<Float> type
}
```

```
fun takeFunction(x: KFunction1<Int, Float>) { ... }

fun main() {
    val result = buildList {
        takeFunction(::get)
    } // result has the List<Float> type
}
```

At the end of the analysis, builder inference considers all collected type information and tries to merge it into the resulting type. See the example.

```
val result = buildList { // Inferring postponed type variable E
    // Considering E is Number or a subtype of Number
    val n: Number? = getOrNull(0)
    // Considering E is Int or a supertype of Int
    add(1)
    // E gets inferred into Int
} // result has the List<Int> type
```

The resulting type is the most specific type that corresponds to the type information collected during the analysis. If the given type information is contradictory and cannot be merged, the compiler reports an error.

Note that the Kotlin compiler uses builder inference only if regular type inference cannot infer a type argument. This means you can contribute type information outside a builder lambda, and then builder inference analysis is not required. Consider the example:

```
fun someMap() = mutableMapOf<CharSequence, String>()

fun <E> MutableMap<E, String>.f(x: MutableMap<E, String>) { ... }

fun main() {
    val x: Map<in String, String> = buildMap {
        put("", "")
        f(someMap()) // Type mismatch (required String, found CharSequence)
    }
}
```

Here a type mismatch appears because the expected type of the map is specified outside the builder lambda. The compiler analyzes all the statements inside with the fixed receiver type Map<in String, String>.

Null safety

Nullable types and non-null types

Kotlin's type system is aimed at eliminating the danger of null references, also known as The Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java this would be the equivalent of a NullPointerException, or an NPE for short.

The only possible causes of an NPE in Kotlin are:

- An explicit call to throw NullPointerException().
- Usage of the !! operator that is described below.
- Data inconsistency with regard to initialization, such as when:
  - An uninitialized this available in a constructor is passed and used somewhere (a "leaking this").
  - A superclass constructor calls an open member whose implementation in the derived class uses an uninitialized state.
- Java interoperation:
  - Attempts to access a member of a null reference of a platform type;
  - Nullability issues with generic types being used for Java interoperation. For example, a piece of Java code might add null into a Kotlin MutableList<String>, therefore requiring a MutableList<String?> for working with it.
  - Other issues caused by external Java code.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references). For example, a regular variable of type String cannot hold null:

```
fun main() {
//sampleStart
    var a: String = "abc" // Regular initialization means non-null by default
    a = null // compilation error
//sampleEnd
}
```

To allow nulls, you can declare a variable as a nullable string by writing String?:

```
fun main() {
//sampleStart
    var b: String? = "abc" // can be set to null
    b = null // ok
    print(b)
//sampleEnd
}
```

Now, if you call a method or access a property on a, it's guaranteed not to cause an NPE, so you can safely say:

```
val l = a.length
```

But if you want to access the same property on b, that would not be safe, and the compiler reports an error:

```
val l = b.length // error: variable 'b' can be null
```

But you still need to access that property, right? There are a few ways to do so.

Checking for null in conditions

First, you can explicitly check whether b is null, and handle the two options separately:

```
val l = if (b != null) b.length else -1
```

The compiler tracks the information about the check you performed, and allows the call to length inside the if. More complex conditions are supported as well:

```
fun main() {
//sampleStart
```

```kotlin
    val b: String? = "Kotlin"
    if (b != null && b.length > 0) {
        print("String of length ${b.length}")
    } else {
        print("Empty string")
    }
//sampleEnd
}
```

Note that this only works where b is immutable (meaning it is a local variable that is not modified between the check and its usage or it is a member val that has a backing field and is not overridable), because otherwise it could be the case that b changes to null after the check.

Safe calls

Your second option for accessing a property on a nullable variable is using the safe call operator ?.:

```kotlin
fun main() {
//sampleStart
    val a = "Kotlin"
    val b: String? = null
    println(b?.length)
    println(a?.length) // Unnecessary safe call
//sampleEnd
}
```

This returns b.length if b is not null, and null otherwise. The type of this expression is Int?.

Safe calls are useful in chains. For example, Bob is an employee who may be assigned to a department (or not). That department may in turn have another employee as a department head. To obtain the name of Bob's department head (if there is one), you write the following:

```kotlin
bob?.department?.head?.name
```

Such a chain returns null if any of the properties in it is null.

To perform a certain operation only for non-null values, you can use the safe call operator together with let:

```kotlin
fun main() {
//sampleStart
    val listWithNulls: List<String?> = listOf("Kotlin", null)
    for (item in listWithNulls) {
         item?.let { println(it) } // prints Kotlin and ignores null
    }
//sampleEnd
}
```

A safe call can also be placed on the left side of an assignment. Then, if one of the receivers in the safe calls chain is null, the assignment is skipped and the expression on the right is not evaluated at all:

```kotlin
// If either `person` or `person.department` is null, the function is not called:
person?.department?.head = managersPool.getManager()
```

Nullable receiver

Extension functions can be defined on a nullable receiver. This way you can specify behaviour for null values without the need to use null-checking logic at each call-site.

For example, the toString() function is defined on a nullable receiver. It returns the String "null" (as opposed to a null value). This can be helpful in certain situations, for example, logging:

```kotlin
val person: Person? = null
logger.debug(person.toString()) // Logs "null", does not throw an exception
```

If you want your toString() invocation to return a nullable string, use the safe-call operator ?.:

```kotlin
var timestamp: Instant? = null
val isoTimestamp = timestamp?.toString() // Returns a String? object which is `null`
if (isoTimestamp == null) {
   // Handle the case where timestamp was `null`
}
```

Elvis operator

When you have a nullable reference, b, you can say "if b is not null, use it, otherwise use some non-null value":

```kotlin
val l: Int = if (b != null) b.length else -1
```

Instead of writing the complete if expression, you can also express this with the Elvis operator ?::

```kotlin
val l = b?.length ?: -1
```

If the expression to the left of ?: is not null, the Elvis operator returns it, otherwise it returns the expression to the right. Note that the expression on the right-hand side is evaluated only if the left-hand side is null.

Since throw and return are expressions in Kotlin, they can also be used on the right-hand side of the Elvis operator. This can be handy, for example, when checking function arguments:

```kotlin
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

The !! operator

The third option is for NPE-lovers: the not-null assertion operator (!!) converts any value to a non-null type and throws an exception if the value is null. You can write b!!, and this will return a non-null value of b (for example, a String in our example) or throw an NPE if b is null:

```kotlin
val l = b!!.length
```

Thus, if you want an NPE, you can have it, but you have to ask for it explicitly and it won't appear out of the blue.

Safe casts

Regular casts may result in a ClassCastException if the object is not of the target type. Another option is to use safe casts that return null if the attempt was not successful:

```kotlin
val aInt: Int? = a as? Int
```

Collections of a nullable type

If you have a collection of elements of a nullable type and want to filter non-null elements, you can do so by using filterNotNull:

```kotlin
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

What's next?

Learn how to handle nullability in Java and Kotlin.

Equality

In Kotlin there are two types of equality:

- Structural equality (== - a check for equals())
- Referential equality (=== - two references point to the same object)

Structural equality

Structural equality is checked by the == operation and its negated counterpart !=. By convention, an expression like a == b is translated to:

```
a?.equals(b) ?: (b === null)
```

If a is not null, it calls the equals(Any?) function, otherwise (a is null) it checks that b is referentially equal to null.

Note that there's no point in optimizing your code when comparing to null explicitly: a == null will be automatically translated to a === null.

To provide a custom equals check implementation, override the equals(other: Any?): Boolean function. Functions with the same name and other signatures, like equals(other: Foo), don't affect equality checks with the operators == and !=.

Structural equality has nothing to do with comparison defined by the Comparable<...> interface, so only a custom equals(Any?) implementation may affect the behavior of the operator.

Referential equality

Referential equality is checked by the === operation and its negated counterpart !==. a === b evaluates to true if and only if a and b point to the same object. For values represented by primitive types at runtime (for example, Int), the === equality check is equivalent to the == check.

Floating-point numbers equality

When an equality check operands are statically known to be Float or Double (nullable or not), the check follows the IEEE 754 Standard for Floating-Point Arithmetic.

Otherwise, structural equality is used, which disagrees with the standard so that NaN is equal to itself, NaN is considered greater than any other element, including POSITIVE_INFINITY, and -0.0 is not equal to 0.0.

See: Floating-point numbers comparison.

This expressions

To denote the current receiver, you use this expressions:

- In a member of a class, this refers to the current object of that class.
- In an extension function or a function literal with receiver this denotes the receiver parameter that is passed on the left-hand side of a dot.

If this has no qualifiers, it refers to the innermost enclosing scope. To refer to this in other scopes, label qualifiers are used:

Qualified this

To access this from an outer scope (a class, extension function, or labeled function literal with receiver) you write this@label, where @label is a label on the scope this is meant to be from:

```
class A { // implicit label @A
    inner class B { // implicit label @B
        fun Int.foo() { // implicit label @foo
            val a = this@A // A's this
            val b = this@B // B's this
```

```
            val c = this // foo()'s receiver, an Int
            val c1 = this@foo // foo()'s receiver, an Int

            val funLit = lambda@ fun String.() {
                val d = this // funLit's receiver, a String
            }

            val funLit2 = { s: String ->
                // foo()'s receiver, since enclosing lambda expression
                // doesn't have any receiver
                val d1 = this
            }
        }
    }
}
```

Implicit this

When you call a member function on this, you can skip the this. part. If you have a non-member function with the same name, use this with caution because in some cases it can be called instead:

```
fun main() {
//sampleStart
    fun printLine() { println("Top-level function") }

    class A {
        fun printLine() { println("Member function") }

        fun invokePrintLine(omitThis: Boolean = false)  {
            if (omitThis) printLine()
            else this.printLine()
        }
    }

    A().invokePrintLine() // Member function
    A().invokePrintLine(omitThis = true) // Top-level function
//sampleEnd()
}
```

Asynchronous programming techniques

For decades, as developers we are confronted with a problem to solve - how to prevent our applications from blocking. Whether we're developing desktop, mobile, or even server-side applications, we want to avoid having the user wait or what's worse cause bottlenecks that would prevent an application from scaling.

There have been many approaches to solving this problem, including:

- Threading
- Callbacks
- Futures, promises, and others
- Reactive Extensions
- Coroutines

Before explaining what coroutines are, let's briefly review some of the other solutions.

Threading

Threads are by far probably the most well-known approach to avoid applications from blocking.

```
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}
```

```
fun preparePost(): Token {
    // makes a request and consequently blocks the main thread
    return token
}
```

Let's assume in the code above that preparePost is a long-running process and consequently would block the user interface. What we can do is launch it in a separate thread. This would then allow us to avoid the UI from blocking. This is a very common technique, but has a series of drawbacks:

- Threads aren't cheap. Threads require context switches which are costly.
- Threads aren't infinite. The number of threads that can be launched is limited by the underlying operating system. In server-side applications, this could cause a major bottleneck.
- Threads aren't always available. Some platforms, such as JavaScript do not even support threads.
- Threads aren't easy. Debugging threads and avoiding race conditions are common problems we suffer in multi-threaded programming.

Callbacks

With callbacks, the idea is to pass one function as a parameter to another function, and have this one invoked once the process has completed.

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}

fun preparePostAsync(callback: (Token) -> Unit) {
    // make request and return immediately
    // arrange callback to be invoked later
}
```

This in principle feels like a much more elegant solution, but once again has several issues:

- Difficulty of nested callbacks. Usually a function that is used as a callback, often ends up needing its own callback. This leads to a series of nested callbacks which lead to incomprehensible code. The pattern is often referred to as the titled christmas tree (braces represent branches of the tree).
- Error handling is complicated. The nesting model makes error handling and propagation of these somewhat more complicated.

Callbacks are quite common in event-loop architectures such as JavaScript, but even there, generally people have moved away to using other approaches such as promises or reactive extensions.

Futures, promises, and others

The idea behind futures or promises (there are also other terms these can be referred to depending on language/platform), is that when we make a call, we're promised that at some point it will return with an object called a Promise, which can then be operated on.

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token ->
            submitPostAsync(token, item)
        }
        .thenAccept { post ->
            processPost(post)
        }

}

fun preparePostAsync(): Promise<Token> {
    // makes request and returns a promise that is completed later
    return promise
}
```

This approach requires a series of changes in how we program, in particular:

- Different programming model. Similar to callbacks, the programming model moves away from a top-down imperative approach to a compositional model with

chained calls. Traditional program structures such as loops, exception handling, etc. usually are no longer valid in this model.

- Different APIs. Usually there's a need to learn a completely new API such as thenCompose or thenAccept, which can also vary across platforms.
- Specific return type. The return type moves away from the actual data that we need and instead returns a new type Promise which has to be introspected.
- Error handling can be complicated. The propagation and chaining of errors aren't always straightforward.

Reactive extensions

Reactive Extensions (Rx) were introduced to C# by Erik Meijer. While it was definitely used on the .NET platform it really didn't reach mainstream adoption until Netflix ported it over to Java, naming it RxJava. From then on, numerous ports have been provided for a variety of platforms including JavaScript (RxJS).

The idea behind Rx is to move towards what's called observable streams whereby we now think of data as streams (infinite amounts of data) and these streams can be observed. In practical terms, Rx is simply the Observer Pattern with a series of extensions which allow us to operate on the data.

In approach it's quite similar to Futures, but one can think of a Future as returning a discrete element, whereas Rx returns a stream. However, similar to the previous, it also introduces a complete new way of thinking about our programming model, famously phrased as

"everything is a stream, and it's observable"

This implies a different way to approach problems and quite a significant shift from what we're used to when writing synchronous code. One benefit as opposed to Futures is that given it's ported to so many platforms, generally we can find a consistent API experience no matter what we use, be it C#, Java, JavaScript, or any other language where Rx is available.

In addition, Rx does introduce a somewhat nicer approach to error handling.

Coroutines

Kotlin's approach to working with asynchronous code is using coroutines, which is the idea of suspendable computations, i.e. the idea that a function can suspend its execution at some point and resume later on.

One of the benefits however of coroutines is that when it comes to the developer, writing non-blocking code is essentially the same as writing blocking code. The programming model in itself doesn't really change.

Take for instance the following code:

```
fun postItem(item: Item) {
    launch {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

suspend fun preparePost(): Token {
    // makes a request and suspends the coroutine
    return suspendCoroutine { /* ... */ }
}
```

This code will launch a long-running operation without blocking the main thread. The preparePost is what's called a suspendable function, thus the keyword suspend prefixing it. What this means as stated above, is that the function will execute, pause execution and resume at some point in time.

- The function signature remains exactly the same. The only difference is suspend being added to it. The return type however is the type we want to be returned.
- The code is still written as if we were writing synchronous code, top-down, without the need of any special syntax, beyond the use of a function called launch which essentially kicks off the coroutine (covered in other tutorials).
- The programming model and APIs remain the same. We can continue to use loops, exception handling, etc. and there's no need to learn a complete set of new APIs.
- It is platform independent. Whether we're targeting JVM, JavaScript or any other platform, the code we write is the same. Under the covers the compiler takes care of adapting it to each platform.

Coroutines are not a new concept, let alone invented by Kotlin. They've been around for decades and are popular in some other programming languages such as Go. What is important to note though is that the way they're implemented in Kotlin, most of the functionality is delegated to libraries. In fact, beyond the suspend keyword, no other keywords are added to the language. This is somewhat different from languages such as C# that have async and await as part of the syntax.

With Kotlin, these are just library functions.

For more information, see the Coroutines reference.

Coroutines

Asynchronous or non-blocking programming is an important part of the development landscape. When creating server-side, desktop, or mobile applications, it's important to provide an experience that is not only fluid from the user's perspective, but also scalable when needed.

Kotlin solves this problem in a flexible way by providing coroutine support at the language level and delegating most of the functionality to libraries.

In addition to opening the doors to asynchronous programming, coroutines also provide a wealth of other possibilities, such as concurrency and actors.

How to start

New to Kotlin? Take a look at the Getting started page.

Documentation

- Coroutines guide
- Basics
- Channels
- Coroutine context and dispatchers
- Shared mutable state and concurrency
- Asynchronous flow

Tutorials

- Asynchronous programming techniques
- Introduction to coroutines and channels
- Debug coroutines using IntelliJ IDEA
- Debug Kotlin Flow using IntelliJ IDEA – tutorial
- Testing Kotlin coroutines on Android

Sample projects

- kotlinx.coroutines examples and sources
- KotlinConf app

Annotations

Annotations are means of attaching metadata to code. To declare an annotation, put the annotation modifier in front of a class:

```
annotation class Fancy
```

Additional attributes of the annotation can be specified by annotating the annotation class with meta-annotations:

- @Target specifies the possible kinds of elements which can be annotated with the annotation (such as classes, functions, properties, and expressions);
- @Retention specifies whether the annotation is stored in the compiled class files and whether it's visible through reflection at runtime (by default, both are true);

- @Repeatable allows using the same annotation on a single element multiple times;
- @MustBeDocumented specifies that the annotation is part of the public API and should be included in the class or method signature shown in the generated API documentation.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.TYPE_PARAMETER, AnnotationTarget.VALUE_PARAMETER,
        AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

Usage

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
        return (@Fancy 1)
    }
}
```

If you need to annotate the primary constructor of a class, you need to add the constructor keyword to the constructor declaration, and add the annotations before it:

```
class Foo @Inject constructor(dependency: MyDependency) { ... }
```

You can also annotate property accessors:

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

Constructors

Annotations can have constructors that take parameters.

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

Allowed parameter types are:

- Types that correspond to Java primitive types (Int, Long etc.)
- Strings
- Classes (Foo::class)
- Enums
- Other annotations
- Arrays of the types listed above

Annotation parameters cannot have nullable types, because the JVM does not support storing null as a value of an annotation attribute.

If an annotation is used as a parameter of another annotation, its name is not prefixed with the @ character:

```
annotation class ReplaceWith(val expression: String)

annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === other"))
```

If you need to specify a class as an argument of an annotation, use a Kotlin class (KClass). The Kotlin compiler will automatically convert it to a Java class, so that the Java code can access the annotations and arguments normally.

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any>)

@Ann(String::class, Int::class) class MyClass
```

Instantiation

In Java, an annotation type is a form of an interface, so you can implement it and use an instance. As an alternative to this mechanism, Kotlin lets you call a constructor of an annotation class in arbitrary code and similarly use the resulting instance.

```
annotation class InfoMarker(val info: String)

fun processInfo(marker: InfoMarker): Unit = TODO()

fun main(args: Array<String>) {
    if (args.isNotEmpty())
        processInfo(getAnnotationReflective(args))
    else
        processInfo(InfoMarker("default"))
}
```

Learn more about instantiation of annotation classes in this KEEP.

Lambdas

Annotations can also be used on lambdas. They will be applied to the invoke() method into which the body of the lambda is generated. This is useful for frameworks like Quasar, which uses annotations for concurrency control.

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

Annotation use-site targets

When you're annotating a property or a primary constructor parameter, there are multiple Java elements which are generated from the corresponding Kotlin element, and therefore multiple possible locations for the annotation in the generated Java bytecode. To specify how exactly the annotation should be generated, use the following syntax:

```
class Example(@field:Ann val foo,    // annotate Java field
              @get:Ann val bar,      // annotate Java getter
              @param:Ann val quux)   // annotate Java constructor parameter
```

The same syntax can be used to annotate the entire file. To do this, put an annotation with the target file at the top level of a file, before the package directive or before all imports if the file is in the default package:

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

If you have multiple annotations with the same target, you can avoid repeating the target by adding brackets after the target and putting all the annotations inside the brackets:

```
class Example {
     @set:[Inject VisibleForTesting]
     var collaborator: Collaborator
```

```
}
```

The full list of supported use-site targets is:

- file
- property (annotations with this target are not visible to Java)
- field
- get (property getter)
- set (property setter)
- receiver (receiver parameter of an extension function or property)
- param (constructor parameter)
- setparam (property setter parameter)
- delegate (the field storing the delegate instance for a delegated property)

To annotate the receiver parameter of an extension function, use the following syntax:

```
fun @receiver:Fancy String.myExtension() { ... }
```

If you don't specify a use-site target, the target is chosen according to the @Target annotation of the annotation being used. If there are multiple applicable targets, the first applicable target from the following list is used:

- param
- property
- field

Java annotations

Java annotations are 100% compatible with Kotlin:

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
import org.junit.rules.*

class Tests {
    // apply @Rule annotation to property getter
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

Since the order of parameters for an annotation written in Java is not defined, you can't use a regular function call syntax for passing the arguments. Instead, you need to use the named argument syntax:

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

Just like in Java, a special case is the value parameter; its value can be specified without an explicit name:

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

Arrays as annotation parameters

If the value argument in Java has an array type, it becomes a vararg parameter in Kotlin:

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

For other arguments that have an array type, you need to use the array literal syntax or arrayOf(...):

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```
@AnnWithArrayMethod(names = ["abc", "foo", "bar"])
class C
```

Accessing properties of an annotation instance

Values of an annotation instance are exposed as properties to Kotlin code:

```
// Java
public @interface Ann {
    int value();
}
```

```
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

Ability to not generate JVM 1.8+ annotation targets

If a Kotlin annotation has TYPE among its Kotlin targets, the annotation maps to java.lang.annotation.ElementType.TYPE_USE in its list of Java annotation targets. This is just like how the TYPE_PARAMETER Kotlin target maps to the java.lang.annotation.ElementType.TYPE_PARAMETER Java target. This is an issue for Android clients with API levels less than 26, which don't have these targets in the API.

To avoid generating the TYPE_USE and TYPE_PARAMETER annotation targets, use the new compiler argument -Xno-new-java-annotation-targets.

Repeatable annotations

Just like in Java, Kotlin has repeatable annotations, which can be applied to a single code element multiple times. To make your annotation repeatable, mark its declaration with the @kotlin.annotation.Repeatable meta-annotation. This will make it repeatable both in Kotlin and Java. Java repeatable annotations are also supported from the Kotlin side.

The main difference with the scheme used in Java is the absence of a containing annotation, which the Kotlin compiler generates automatically with a predefined name. For an annotation in the example below, it will generate the containing annotation @Tag.Container:

```kotlin
@Repeatable
annotation class Tag(val name: String)

// The compiler generates the @Tag.Container containing annotation
```

You can set a custom name for a containing annotation by applying the @kotlin.jvm.JvmRepeatable meta-annotation and passing an explicitly declared containing annotation class as an argument:

```kotlin
@JvmRepeatable(Tags::class)
annotation class Tag(val name: String)

annotation class Tags(val value: Array<Tag>)
```

To extract Kotlin or Java repeatable annotations via reflection, use the KAnnotatedElement.findAnnotations() function.

Learn more about Kotlin repeatable annotations in this KEEP.

Destructuring declarations

Sometimes it is convenient to destructure an object into a number of variables, for example:

```kotlin
val (name, age) = person
```

This syntax is called a destructuring declaration. A destructuring declaration creates multiple variables at once. You have declared two new variables: name and age, and can use them independently:

```kotlin
println(name)
println(age)
```

A destructuring declaration is compiled down to the following code:

```kotlin
val name = person.component1()
val age = person.component2()
```

The component1() and component2() functions are another example of the principle of conventions widely used in Kotlin (see operators like + and *, for-loops as an example). Anything can be on the right-hand side of a destructuring declaration, as long as the required number of component functions can be called on it. And, of course, there can be component3() and component4() and so on.

> The componentN() functions need to be marked with the operator keyword to allow using them in a destructuring declaration.

Destructuring declarations also work in for-loops:

```kotlin
for ((a, b) in collection) { ... }
```

Variables a and b get the values returned by component1() and component2() called on elements of the collection.

Example: returning two values from a function

Assume that you need to return two things from a function - for example, a result object and a status of some sort. A compact way of doing this in Kotlin is to declare a data class and return its instance:

```kotlin
data class Result(val result: Int, val status: Status)
fun function(...): Result {
    // computations

    return Result(result, status)
}
```

```
// Now, to use this function:
val (result, status) = function(...)
```

Since data classes automatically declare componentN() functions, destructuring declarations work here.

> You could also use the standard class Pair and have function() return Pair<Int, Status>, but it's often better to have your data named properly.

Example: destructuring declarations and maps

Probably the nicest way to traverse a map is this:

```
for ((key, value) in map) {
   // do something with the key and the value
}
```

To make this work, you should

- Present the map as a sequence of values by providing an iterator() function.
- Present each of the elements as a pair by providing functions component1() and component2().

And indeed, the standard library provides such extensions:

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> = entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

So you can freely use destructuring declarations in for-loops with maps (as well as collections of data class instances or similar).

Underscore for unused variables

If you don't need a variable in the destructuring declaration, you can place an underscore instead of its name:

```
val (_, status) = getResult()
```

The componentN() operator functions are not called for the components that are skipped in this way.

Destructuring in lambdas

You can use the destructuring declarations syntax for lambda parameters. If a lambda has a parameter of the Pair type (or Map.Entry, or any other type that has the appropriate componentN functions), you can introduce several new parameters instead of one by putting them in parentheses:

```
map.mapValues { entry -> "${entry.value}!" }
map.mapValues { (key, value) -> "$value!" }
```

Note the difference between declaring two parameters and declaring a destructuring pair instead of a parameter:

```
{ a -> ... } // one parameter
{ a, b -> ... } // two parameters
{ (a, b) -> ... } // a destructured pair
{ (a, b), c -> ... } // a destructured pair and another parameter
```

If a component of the destructured parameter is unused, you can replace it with the underscore to avoid inventing its name:

```
map.mapValues { (_, value) -> "$value!" }
```

You can specify the type for the whole destructured parameter or for a specific component separately:

```kotlin
map.mapValues { (_, value): Map.Entry<Int, String> -> "$value!" }

map.mapValues { (_, value: String) -> "$value!" }
```

Reflection

Reflection is a set of language and library features that allows you to introspect the structure of your program at runtime. Functions and properties are first-class citizens in Kotlin, and the ability to introspect them (for example, learning the name or the type of a property or function at runtime) is essential when using a functional or reactive style.

> Kotlin/JS provides limited support for reflection features. Learn more about reflection in Kotlin/JS.

JVM dependency

On the JVM platform, the Kotlin compiler distribution includes the runtime component required for using the reflection features as a separate artifact, kotlin-reflect.jar. This is done to reduce the required size of the runtime library for applications that do not use reflection features.

To use reflection in a Gradle or Maven project, add the dependency on kotlin-reflect:

- In Gradle:

  Kotlin

  ```kotlin
  dependencies {
      implementation("org.jetbrains.kotlin:kotlin-reflect:1.9.0")
  }
  ```

  Groovy

  ```groovy
  dependencies {
      implementation "org.jetbrains.kotlin:kotlin-reflect:1.9.0"
  }
  ```

- In Maven:

  ```xml
  <dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-reflect</artifactId>
    </dependency>
  </dependencies>
  ```

If you don't use Gradle or Maven, make sure you have kotlin-reflect.jar in the classpath of your project. In other supported cases (IntelliJ IDEA projects that use the command-line compiler or Ant), it is added by default. In the command-line compiler and Ant, you can use the -no-reflect compiler option to exclude kotlin-reflect.jar from the classpath.

Class references

The most basic reflection feature is getting the runtime reference to a Kotlin class. To obtain the reference to a statically known Kotlin class, you can use the class literal syntax:

```kotlin
val c = MyClass::class
```

The reference is a KClass type value.

On JVM: a Kotlin class reference is not the same as a Java class reference. To obtain a Java class reference, use the .java property on a KClass instance.

Bound class references

You can get the reference to the class of a specific object with the same ::class syntax by using the object as a receiver:

```
val widget: Widget = ...
assert(widget is GoodWidget) { "Bad widget: ${widget::class.qualifiedName}" }
```

You will obtain the reference to the exact class of an object, for example, GoodWidget or BadWidget, regardless of the type of the receiver expression (Widget).

Callable references

References to functions, properties, and constructors can also be called or used as instances of function types.

The common supertype for all callable references is KCallable<out R>, where R is the return value type. It is the property type for properties, and the constructed type for constructors.

Function references

When you have a named function declared as below, you can call it directly (isOdd(5)):

```
fun isOdd(x: Int) = x % 2 != 0
```

Alternatively, you can use the function as a function type value, that is, pass it to another function. To do so, use the :: operator:

```
fun isOdd(x: Int) = x % 2 != 0

fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd))
//sampleEnd
}
```

Here ::isOdd is a value of function type (Int) -> Boolean.

Function references belong to one of the KFunction<out R> subtypes, depending on the parameter count. For instance, KFunction3<T1, T2, T3, R>.

:: can be used with overloaded functions when the expected type is known from the context. For example:

```
fun main() {
//sampleStart
    fun isOdd(x: Int) = x % 2 != 0
    fun isOdd(s: String) = s == "brillig" || s == "slithy" || s == "tove"

    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd)) // refers to isOdd(x: Int)
//sampleEnd
}
```

Alternatively, you can provide the necessary context by storing the method reference in a variable with an explicitly specified type:

```
val predicate: (String) -> Boolean = ::isOdd   // refers to isOdd(x: String)
```

If you need to use a member of a class or an extension function, it needs to be qualified: String::toCharArray.

Even if you initialize a variable with a reference to an extension function, the inferred function type will have no receiver, but it will have an additional parameter accepting a receiver object. To have a function type with a receiver instead, specify the type explicitly:

```
val isEmptyStringList: List<String>.() -> Boolean = List<String>::isEmpty
```

Example: function composition

Consider the following function:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

It returns a composition of two functions passed to it: compose(f, g) = f(g(*)). You can apply this function to callable references:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}

fun isOdd(x: Int) = x % 2 != 0

fun main() {
//sampleStart
    fun length(s: String) = s.length

    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")

    println(strings.filter(oddLength))
//sampleEnd
}
```

Property references

To access properties as first-class objects in Kotlin, use the :: operator:

```
val x = 1

fun main() {
    println(::x.get())
    println(::x.name)
}
```

The expression ::x evaluates to a KProperty0<Int> type property object. You can read its value using get() or retrieve the property name using the name property. For more information, see the docs on the KProperty class.

For a mutable property such as var y = 1, ::y returns a value with the KMutableProperty0<Int> type which has a set() method:

```
var y = 1

fun main() {
    ::y.set(2)
    println(y)
}
```

A property reference can be used where a function with a single generic parameter is expected:

```
fun main() {
//sampleStart
    val strs = listOf("a", "bc", "def")
    println(strs.map(String::length))
//sampleEnd
}
```

To access a property that is a member of a class, qualify it as follows:

```
fun main() {
//sampleStart
    class A(val p: Int)
    val prop = A::p
    println(prop.get(A(1)))
//sampleEnd
}
```

For an extension property:

```kotlin
val String.lastChar: Char
    get() = this[length - 1]

fun main() {
    println(String::lastChar.get("abc"))
}
```

Interoperability with Java reflection

On the JVM platform, the standard library contains extensions for reflection classes that provide a mapping to and from Java reflection objects (see package kotlin.reflect.jvm). For example, to find a backing field or a Java method that serves as a getter for a Kotlin property, you can write something like this:

```kotlin
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main() {
    println(A::p.javaGetter) // prints "public final int A.getP()"
    println(A::p.javaField)  // prints "private final int A.p"
}
```

To get the Kotlin class that corresponds to a Java class, use the .kotlin extension property:

```kotlin
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

Constructor references

Constructors can be referenced just like methods and properties. You can use them wherever the program expects a function type object that takes the same parameters as the constructor and returns an object of the appropriate type. Constructors are referenced by using the :: operator and adding the class name. Consider the following function that expects a function parameter with no parameters and return type Foo:

```kotlin
class Foo

fun function(factory: () -> Foo) {
    val x: Foo = factory()
}
```

Using ::Foo, the zero-argument constructor of the class Foo, you can call it like this:

```kotlin
function(::Foo)
```

Callable references to constructors are typed as one of the KFunction<out R> subtypes depending on the parameter count.

Bound function and property references

You can refer to an instance method of a particular object:

```kotlin
fun main() {
//sampleStart
    val numberRegex = "\\d+".toRegex()
    println(numberRegex.matches("29"))

    val isNumber = numberRegex::matches
    println(isNumber("29"))
//sampleEnd
}
```

Instead of calling the method matches directly, the example uses a reference to it. Such a reference is bound to its receiver. It can be called directly (like in the example above) or used whenever a function type expression is expected:

```kotlin
fun main() {
//sampleStart
    val numberRegex = "\\d+".toRegex()
    val strings = listOf("abc", "124", "a70")
    println(strings.filter(numberRegex::matches))
```

```
//sampleEnd
}
```

Compare the types of the bound and the unbound references. The bound callable reference has its receiver "attached" to it, so the type of the receiver is no longer a parameter:

```
val isNumber: (CharSequence) -> Boolean = numberRegex::matches

val matches: (Regex, CharSequence) -> Boolean = Regex::matches
```

A property reference can be bound as well:

```
fun main() {
//sampleStart
    val prop = "abc"::length
    println(prop.get())
//sampleEnd
}
```

You don't need to specify this as the receiver: this::foo and ::foo are equivalent.

Bound constructor references

A bound callable reference to a constructor of an inner class can be obtained by providing an instance of the outer class:

```
class Outer {
    inner class Inner
}

val o = Outer()
val boundInnerCtor = o::Inner
```

Get started with Kotlin Multiplatform for mobile

> Kotlin Multiplatform is in Beta. It is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make.

Kotlin Multiplatform technology simplifies the development of cross-platform projects. The Kotlin applications will work on different operating systems like iOS, Android, macOS, Windows, Linux, watchOS, and others.

One of the major Kotlin Multiplatform use cases is sharing code between mobile platforms. You can share application logic between iOS and Android apps and write platform-specific code only when you need to implement a native UI or work with platform APIs.

Compose Multiplatform, a JetBrains' declarative UI framework based on Kotlin and Jetpack Compose, gives you the option to push the sharing capabilities of Kotlin Multiplatform beyond application logic. It allows you to implement your user interface once and then use it for all the platforms you target – iOS, Android, desktop, and web.

- Check out our Kotlin Multiplatform Mobile Is in Beta video to learn about the current state and future plans for the technology.
- See how different companies are already using Kotlin for cross-platform app development in production.
- Try Compose Multiplatform, JetBrains' declarative UI framework based on Kotlin and Jetpack Compose to share UIs among iOS, Android, desktop, and web.

Get to know Kotlin Multiplatform and create a mobile app that works on both Android and iOS by completing these steps:

> This tutorial describes how to share application logic between iOS and Android using Kotlin Multiplatform. To learn about the full capabilities of the technology, check out other use cases.

1 Set up an environment for cross-platform mobile development

2 Create your first app that works both on Android and iOS with the IDE

3 Add dependencies to your project

4 Upgrade your app

5 Wrap up your project

Next step

Start by setting up an environment for mobile development.

Proceed to the next part

See also

If you want to convert your existing Android project into a cross-platform app, follow these steps to make it work on iOS:

1 Set up an environment for cross-platform mobile development

2 Complete this tutorial to make your Android app cross-platform

Join the community

- Kotlin Slack: get an invitation and join the #multiplatform channel
- Stack Overflow: subscribe to the "kotlin-multiplatform" tag
- Kotlin YouTube channel: subscribe and watch videos about Kotlin Multiplatform

Set up an environment

Before you create your first application that works on both iOS and Android, you'll need to set up an environment for Kotlin Multiplatform Mobile development.

> To write iOS-specific code and run an iOS application on a simulated or real device, you'll need a Mac with macOS. This cannot be performed on other operating systems, such as Microsoft Windows. This is an Apple requirement.

Install the necessary tools

We recommend that you install the latest stable versions for compatibility and better performance.

Tool	Comments
Android Studio	You will use Android Studio to create your multiplatform applications and run them on simulated or hardware devices.
Xcode	Most of the time, Xcode will work in the background. You will use it to add Swift or Objective-C code to your iOS application. We generally recommend using the latest stable versions for all tools. However, Kotlin/Native sometimes doesn't support the newest Xcode right away. If that's your case, install an earlier version of Xcode.

Tool	Comments
JDK	To check whether it's installed, run the following command in the Android Studio terminal or your command line: `java -version`
Kotlin Multiplatform Mobile plugin	In Android Studio, select Settings/Preferences | Plugins, search Marketplace for Kotlin Multiplatform Mobile, and then install it.
Kotlin plugin	The Kotlin plugin is bundled with each Android Studio release. However, it still needs to be updated to the latest version to avoid compatibility issues. To update the plugin, on the Android Studio welcome screen, select Plugins | Installed. Click Update next to Kotlin. You can also check the Kotlin version in Tools | Kotlin | Configure Kotlin Plugin Updates. The Kotlin plugin should be compatible with the Kotlin Multiplatform Mobile plugin. Refer to the compatibility table.

Check your environment

To make sure everything works as expected, install and run the KDoctor tool:

> KDoctor works on macOS only.

1. In the Android Studio terminal or your command-line tool, run the following command to install the tool using Homebrew:

   ```
   brew install kdoctor
   ```

   If you don't have Homebrew yet, install it or see the KDoctor README for other ways to install it.

2. After the installation is completed, call KDoctor in the console:

   ```
   kdoctor
   ```

3. If KDoctor diagnoses any problems while checking your environment, review the output for issues and possible solutions:

   - Fix any failed checks ([x]). You can find problem descriptions and potential solutions after the * symbol.
   - Check the warnings ([!]) and successful messages ([v]). They may contain useful notes and tips, as well.

   > You may ignore KDoctor's warnings regarding the CocoaPods installation. In your first project, you will use a different iOS framework distribution option.

Possible issues and solutions

Android Studio

Make sure that you have Android Studio installed. You can get it from its official website.

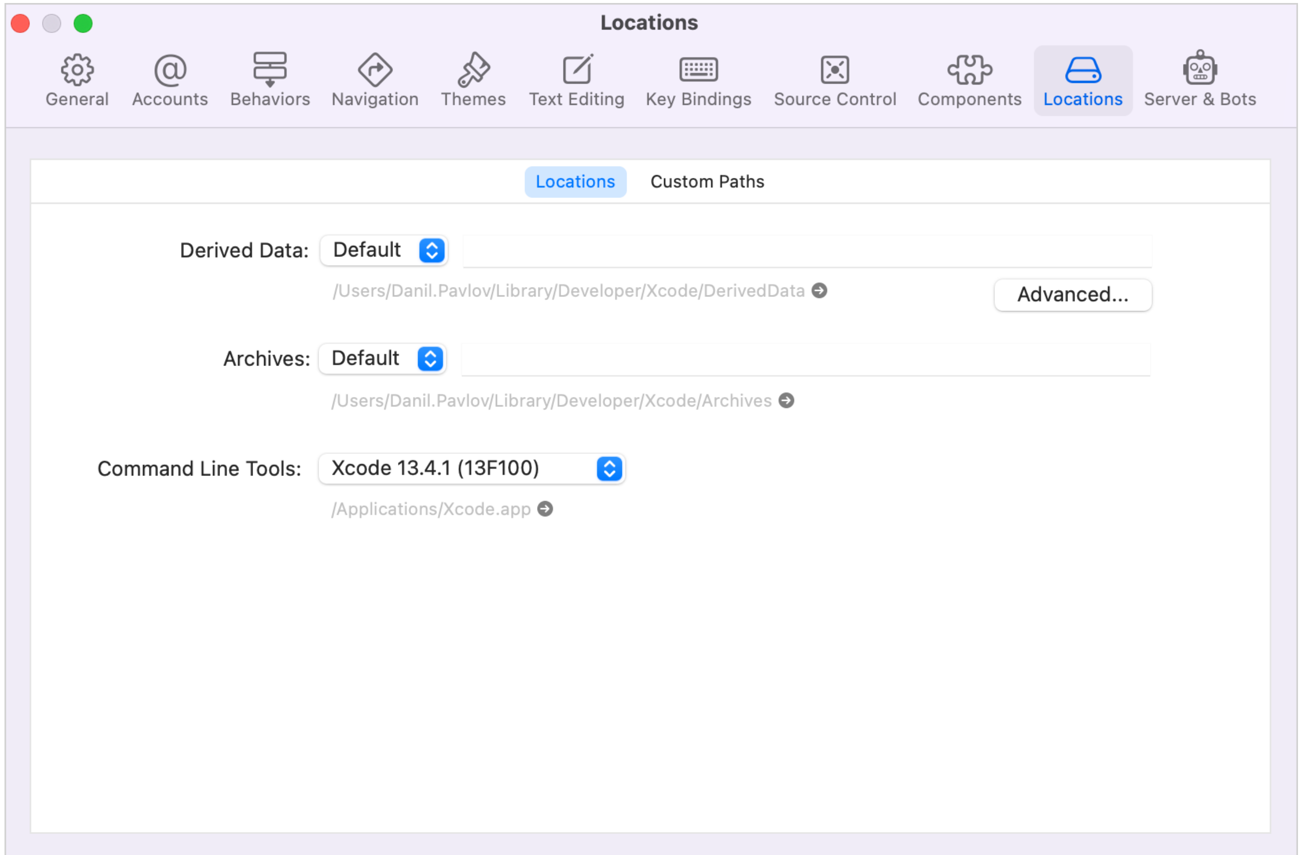
Java and JDK

- Make sure that you have JDK installed. You can get it from its official website.
- Android Studio uses a bundled JDK to execute Gradle tasks. To configure the Gradle JDK in Android Studio, select Settings/Preferences | Build, Execution, Deployment | Build Tools | Gradle.

- You might encounter issues related to JAVA_HOME. This environment variable specifies the location of the Java binary required for Xcode and Gradle. If so, follow KDoctor's tips to fix the issues.

Xcode

- Make sure that you have Xcode installed. You can get it from its official website.
- Launch Xcode in a separate window to accept its license terms and allow it to perform some necessary initial tasks.
- Error: can't grab Xcode schemes. If you encounter an error like this, in Xcode, select Settings/Preferences | Locations. In the Command Line Tools field, select your Xcode.



Xcode schemes

Kotlin plugins

Kotlin Multiplatform Mobile plugin

- Make sure that the Kotlin Mobile Multiplatform plugin is installed and enabled. On the Android Studio welcome screen, select Plugins | Installed. Verify that you have the plugin enabled. If it's not in the Installed list, search Marketplace for it and install the plugin.
- If the plugin is outdated, click Update next to the plugin name. You can do the same in the Settings/Preferences | Tools | Plugins section.
- Check the compatibility of the Kotlin Multiplatform Mobile plugin with your version of Kotlin in the Release details table.

Kotlin plugin

Make sure that the Kotlin plugin is updated to the latest version. To do that, on the Android Studio welcome screen, select Plugins | Installed. Click Update next to Kotlin.

You can also check the Kotlin version in Tools | Kotlin | Configure Kotlin Plugin Updates.

Command line

Make sure you have all the necessary tools installed:

- command not found: brew — install Homebrew.
- command not found: java — install Java.

Next step

In the next part of the tutorial, you'll create your first cross-platform mobile application.

Proceed to the next part

Get help

- Kotlin Slack. Get an invite and join the #multiplatform channel.
- Kotlin issue tracker. Report a new issue.

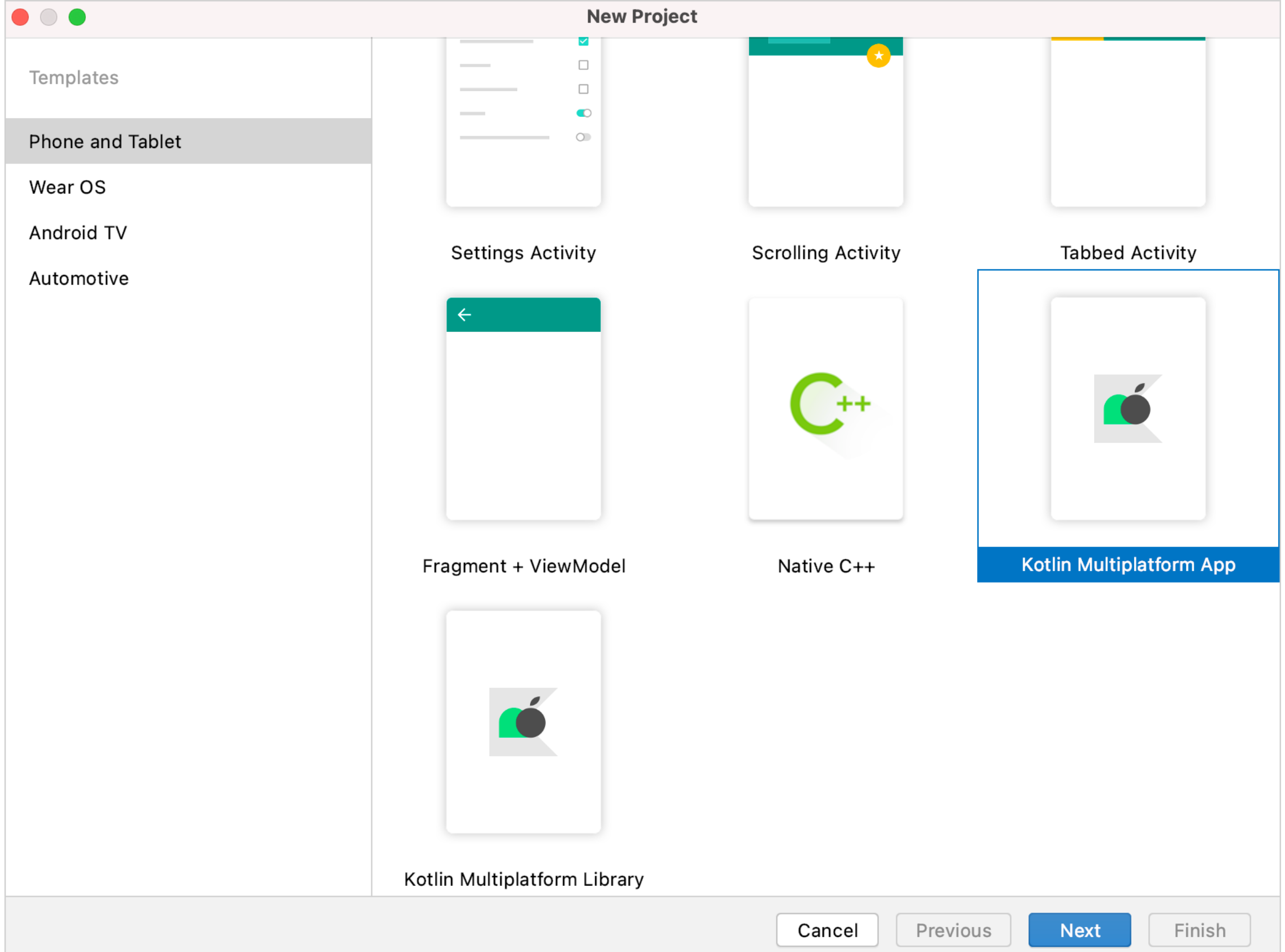
Create your first cross-platform app

Here you will learn how to create and run your first Kotlin Multiplatform application using Android Studio.

Create the project from a template

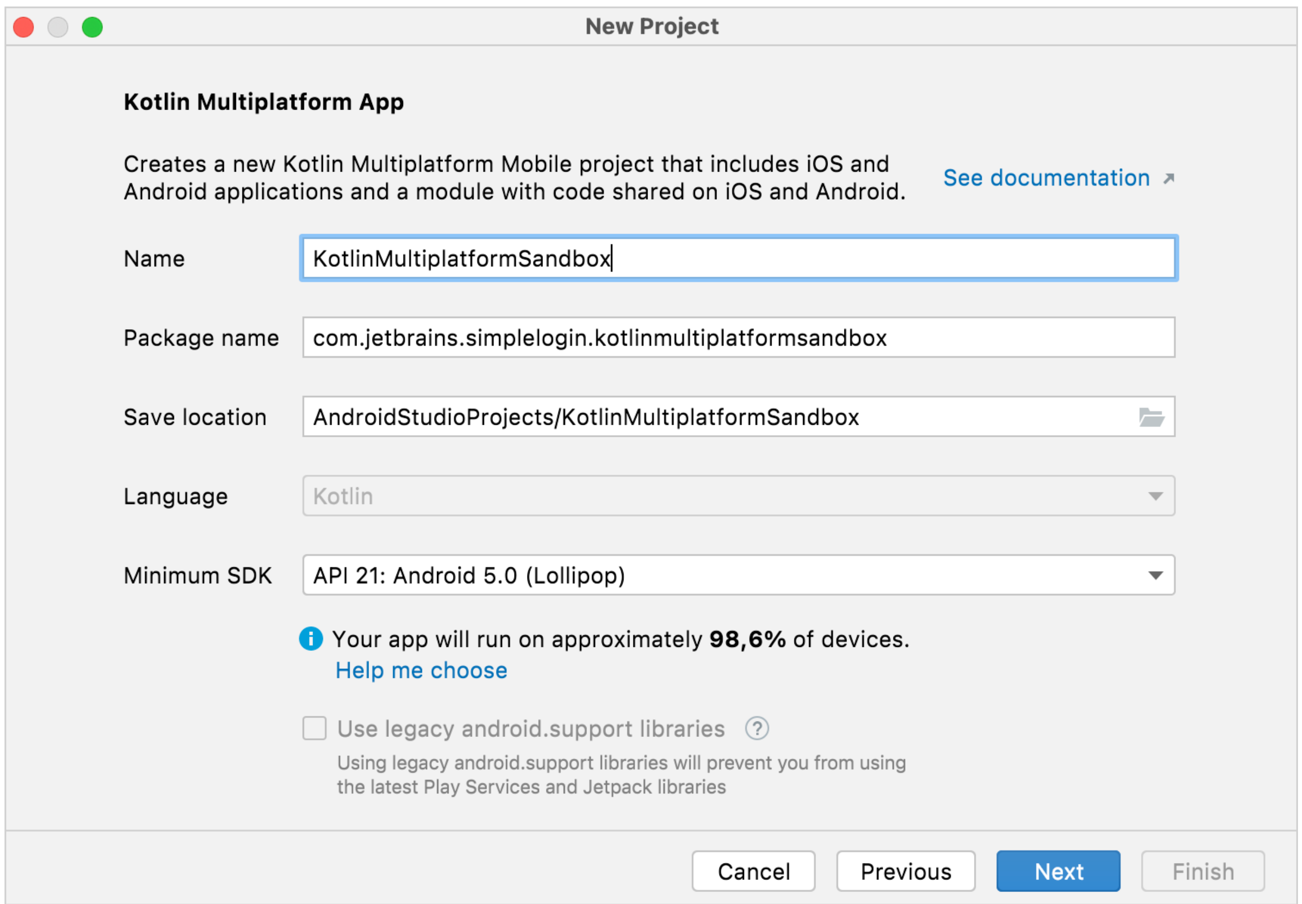
> You can also watch the video version of this tutorial created by Ekaterina Petrova, Kotlin Product Marketing Manager.

1. In Android Studio, select File | New | New Project.
2. Select Kotlin Multiplatform App in the list of project templates, and click Next.



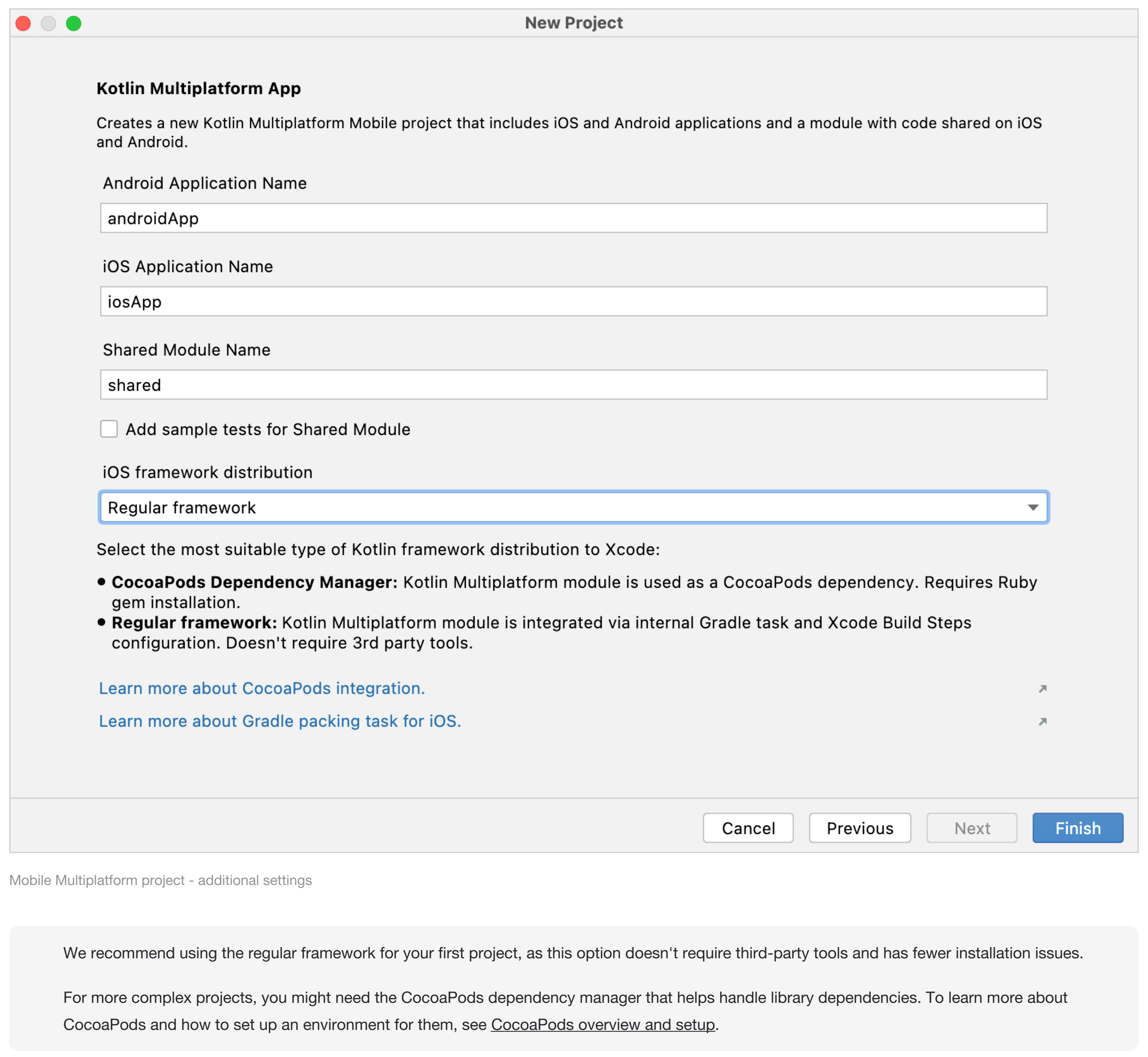
Mobile Multiplatform project template

3. Specify a name for your first application, and click Next.



Mobile Multiplatform project - general settings

4. In the iOS framework distribution list, select the Regular framework option.



Mobile Multiplatform project - additional settings

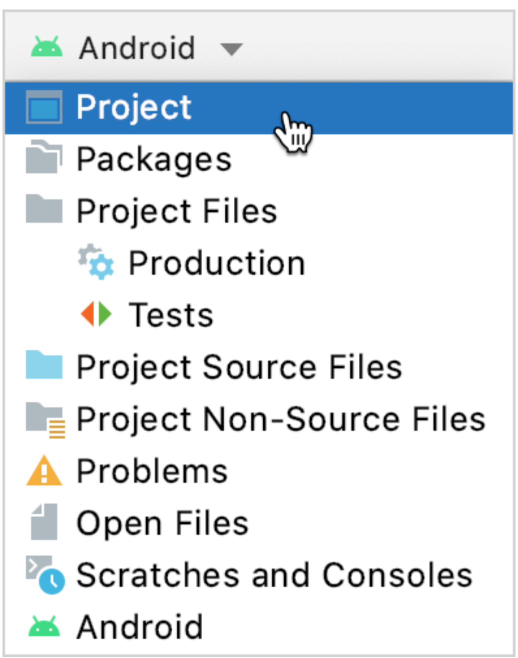
> We recommend using the regular framework for your first project, as this option doesn't require third-party tools and has fewer installation issues.
>
> For more complex projects, you might need the CocoaPods dependency manager that helps handle library dependencies. To learn more about CocoaPods and how to set up an environment for them, see CocoaPods overview and setup.

5. Keep the default names for the application and shared folders. Click Finish.

The project will be set up automatically. It may take some time to download and set up the required components when you do this for the first time.

Examine the project structure

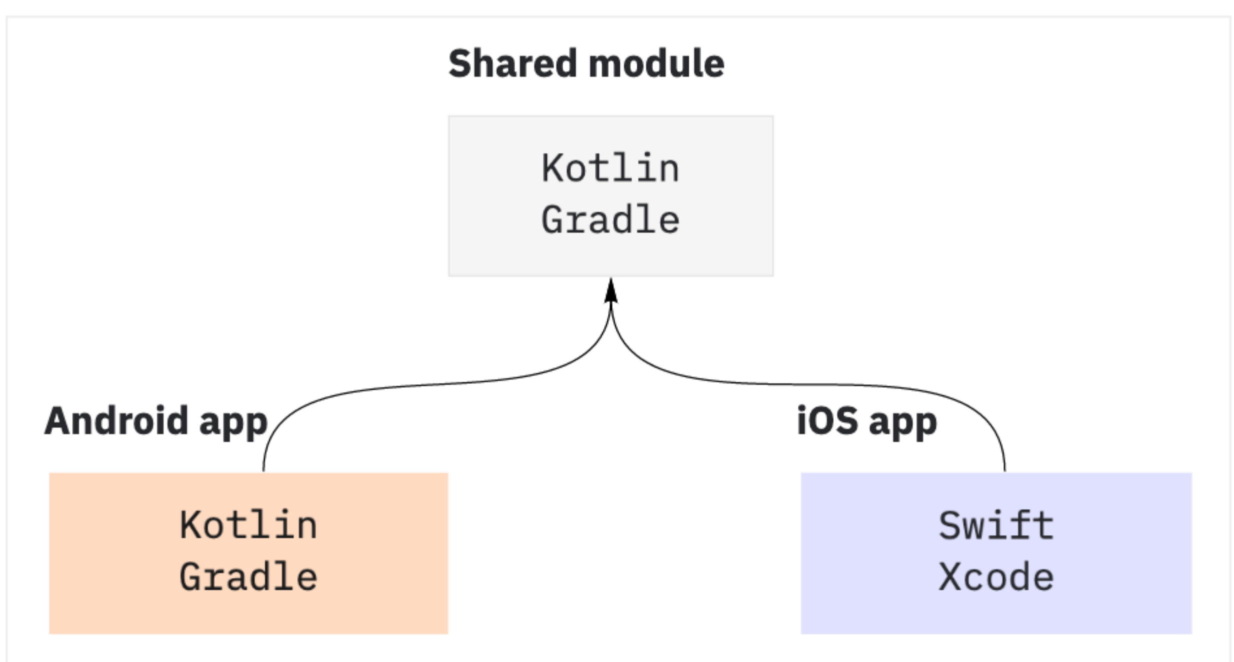
To view the full structure of your mobile multiplatform project, switch the view from Android to Project.



Select the Project view

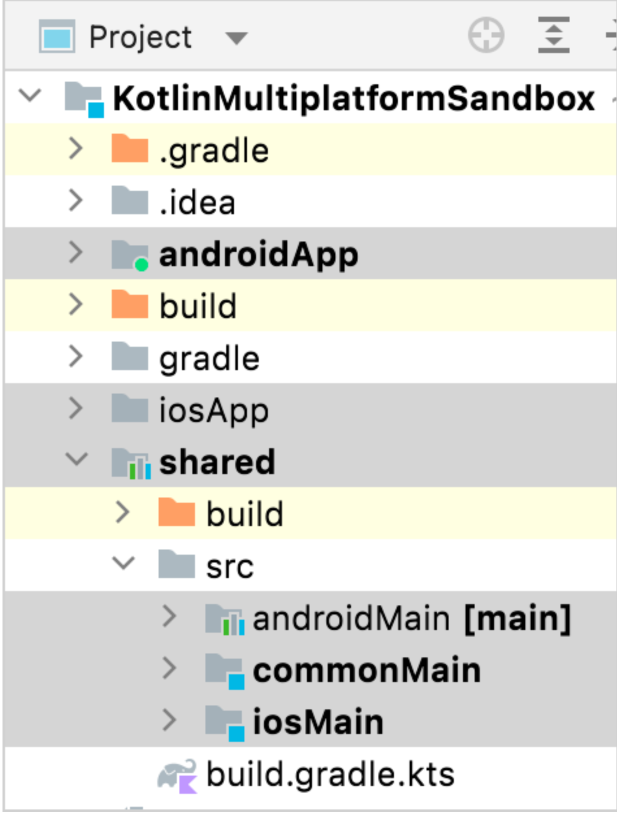
Each Kotlin Multiplatform project includes three modules:

- shared is a Kotlin module that contains the logic common for both Android and iOS applications – the code you share between platforms. It uses Gradle as the build system that helps you automate your build process. The shared module builds into an Android library and an iOS framework.
- androidApp is a Kotlin module that builds into an Android application. It uses Gradle as the build system. The androidApp module depends on and uses the shared module as a regular Android library.
- iosApp is an Xcode project that builds into an iOS application. It depends on and uses the shared module as an iOS framework. The shared module can be used as a regular framework or as a CocoaPods dependency, based on what you've chosen in the previous step in iOS framework distribution. In this tutorial, it's a regular framework dependency.



Basic Multiplatform Mobile project structure

The shared module consists of three source sets: androidMain, commonMain, and iosMain. Source set is a Gradle concept for a number of files logically grouped together where each group has its own dependencies. In Kotlin Multiplatform, different source sets in a shared module can target different platforms.



Source sets and modules structure

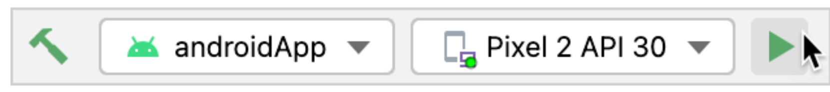
> This is an example structure of a Multiplatform Mobile project that you create with the project wizard in IntelliJ IDEA or Android Studio. Real-life projects can have more complex structures.

Run your application

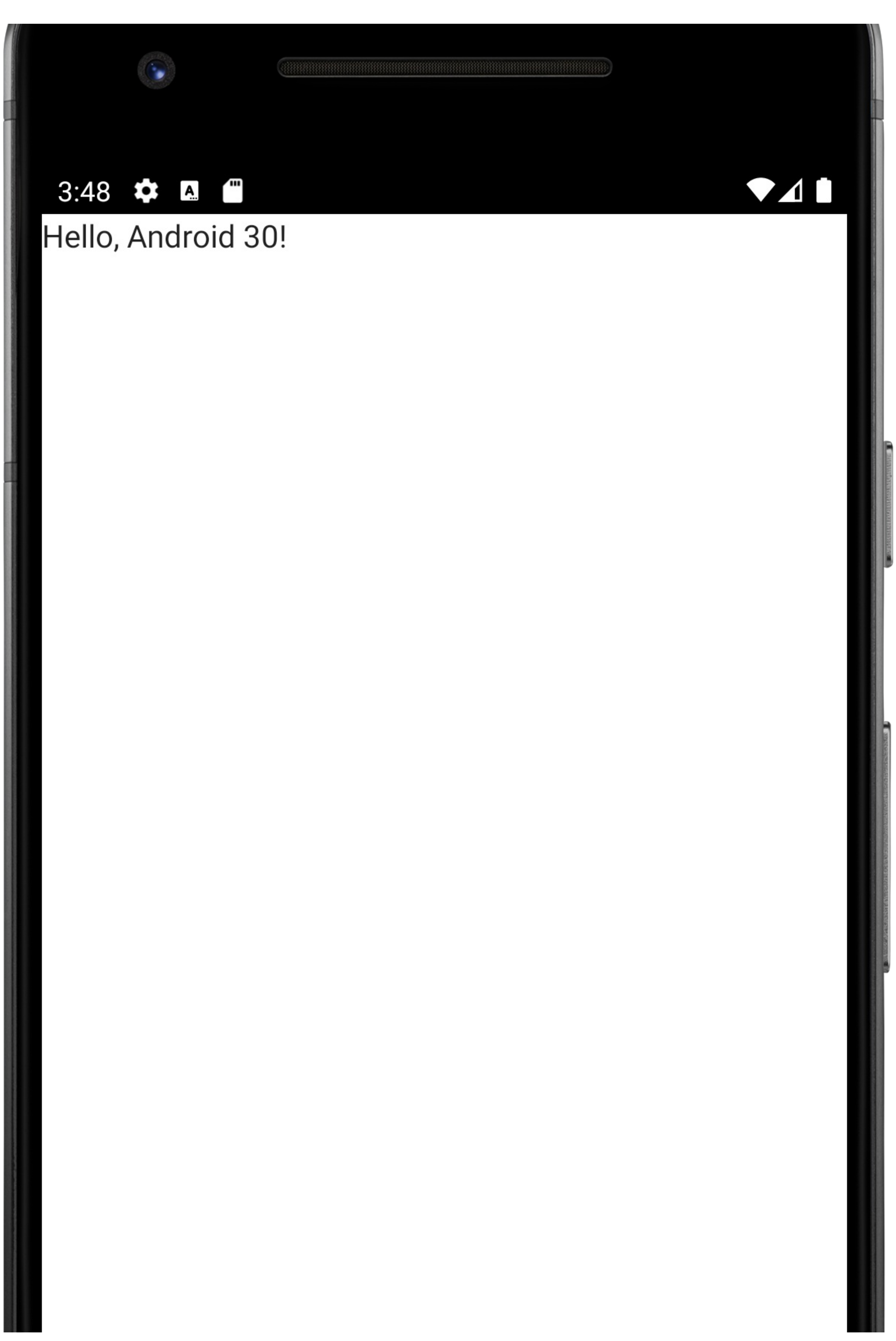
You can run your multiplatform application on Android or iOS.

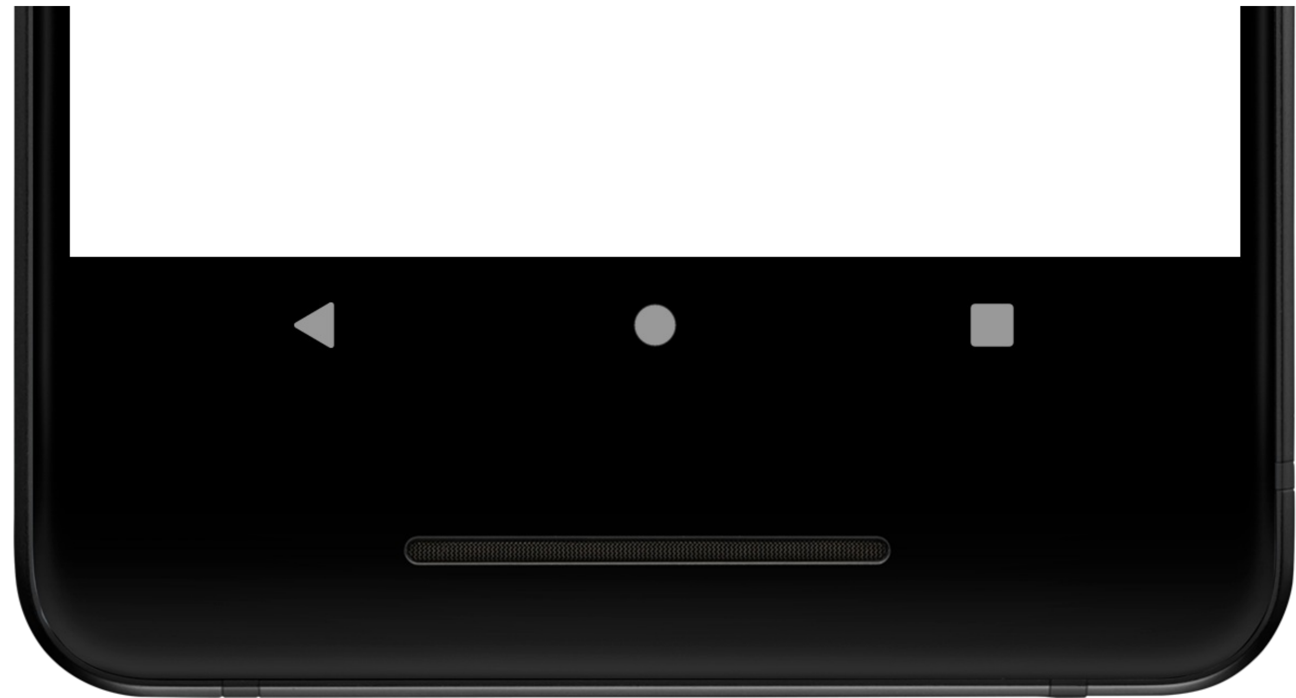
Run your application on Android

1. Create an Android virtual device.
2. In the list of run configurations, select androidApp.
3. Choose your Android virtual device and click Run.



Run multiplatform app on Android


3:48
Hello, Android 30!



First mobile multiplatform app on Android

Run on a different Android simulated device

Learn how to configure the Android Emulator and run your application on a different simulated device.

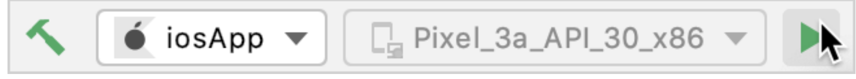
Run on a real Android device

Learn how to configure and connect a hardware device and run your application on it.

Run your application on iOS

1. Launch Xcode in a separate window. The first time you may also need to accept its license terms and allow it to perform some necessary initial tasks.
2. In Android Studio, select iosApp in the list of run configurations and click Run.

   If you don't have an available iOS configuration in the list, add a new iOS simulated device.



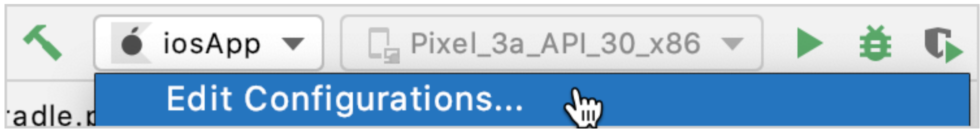
Run multiplatform app on iOS

First mobile multiplatform app on Android

Run on a new iOS simulated device

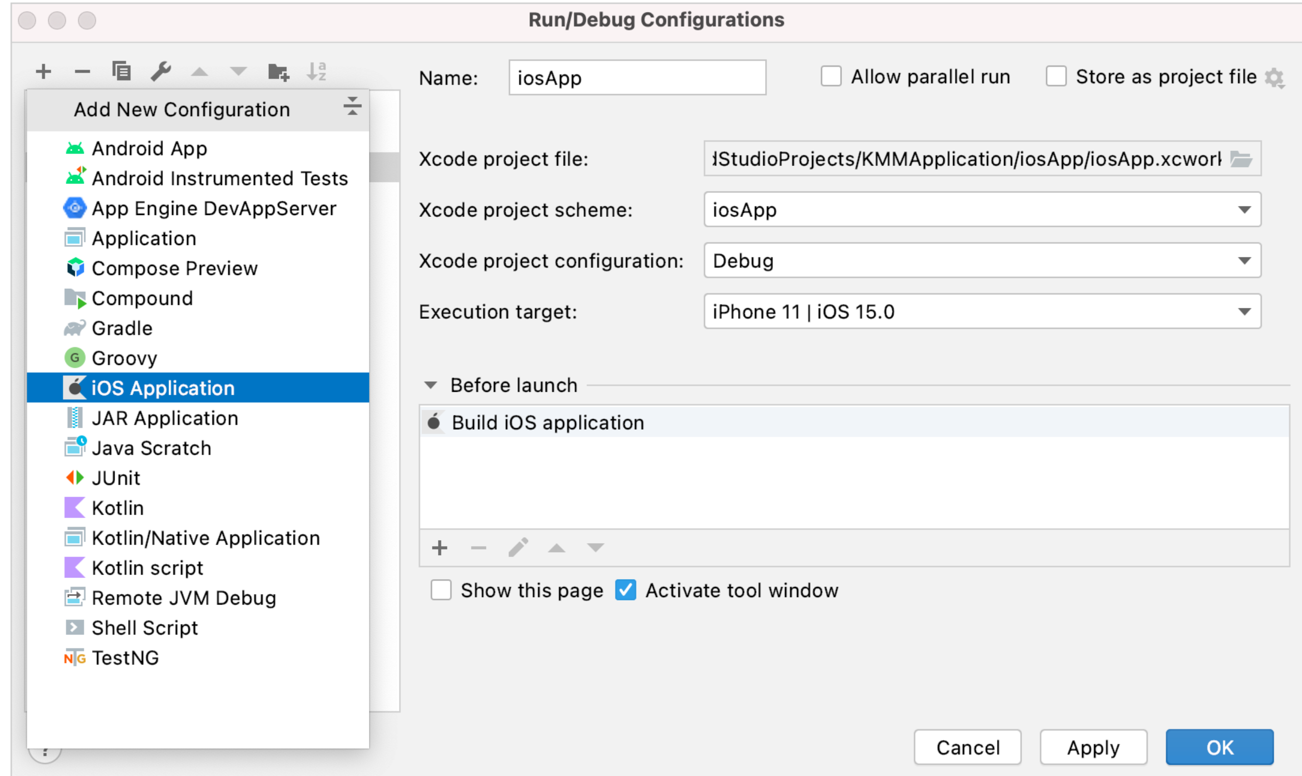
If you want to run your application on a simulated device, you can add a new run configuration.

1. In the list of run configurations, click Edit Configurations.



Edit run configurations

2. Click the + button above the list of configurations and select iOS Application.

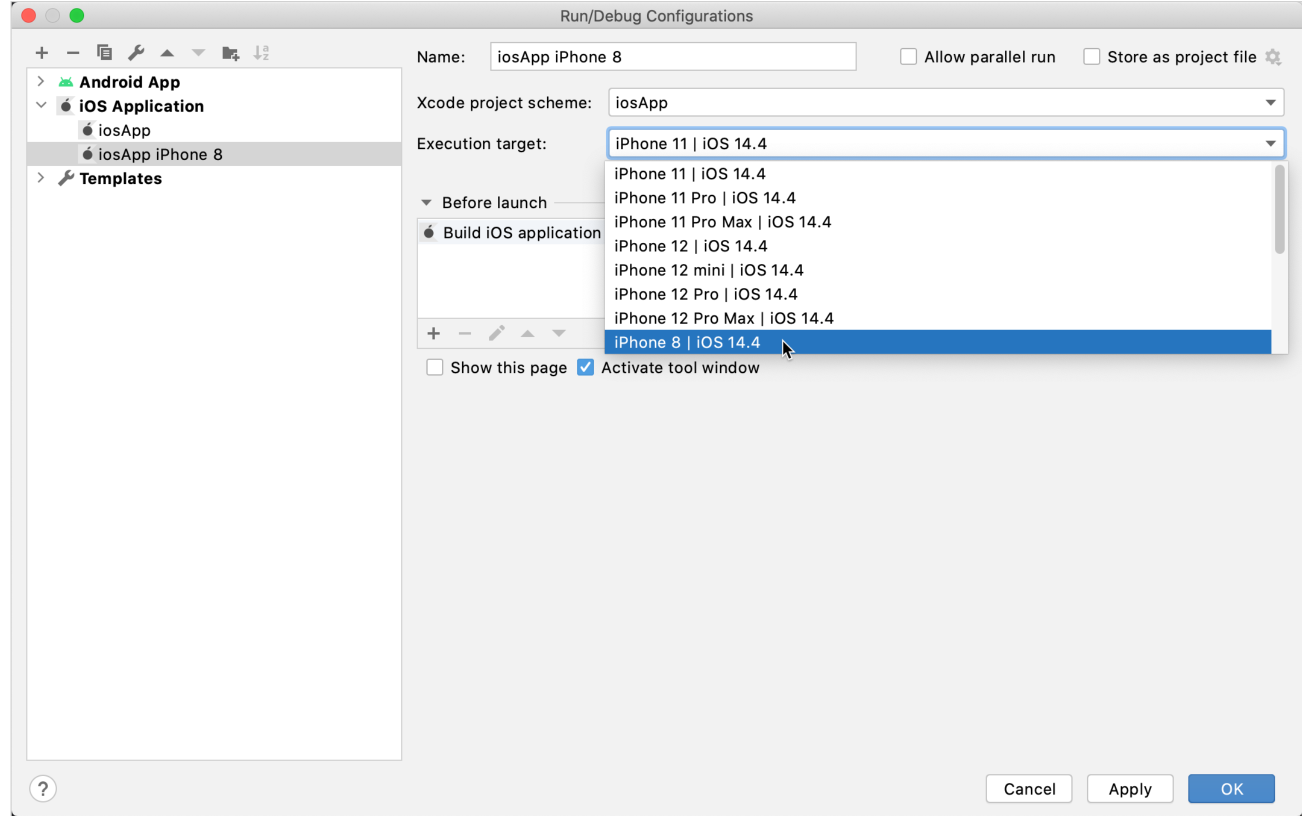


New run configuration for iOS application

3. Name your configuration.

4. Select the Xcode project file. For that, navigate to your project, for example KotlinMultiplatformSandbox, open theiosApp folder and select the .xcodeproj file.

5. In the Execution target list, select a simulated device and click OK.



New run configuration with iOS simulator

6. Click Run to run your application on the new simulated device.

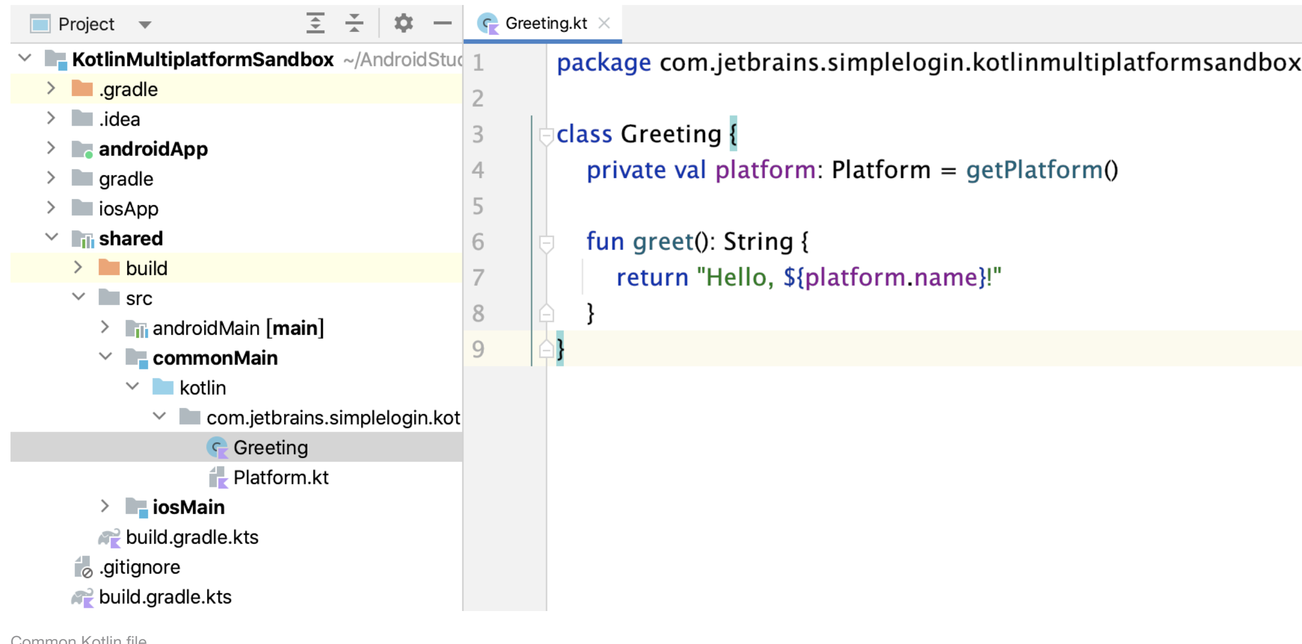
Run on a real iOS device

1. Connect a real iPhone device to Xcode.
2. Make sure to code sign your app. For more information, see the official Apple documentation.
3. Create a run configuration by selecting an iPhone in the Execution target list.
4. Click Run to run your application on the iPhone device.

> If your build fails, follow the workaround described in this issue.

Update your application

1. In shared/src/commonMain/kotlin, open the Greeting.kt file in the project folder. This directory stores the shared code for both Android and iOS. If you make changes to the shared code, you will see them reflected in both applications.

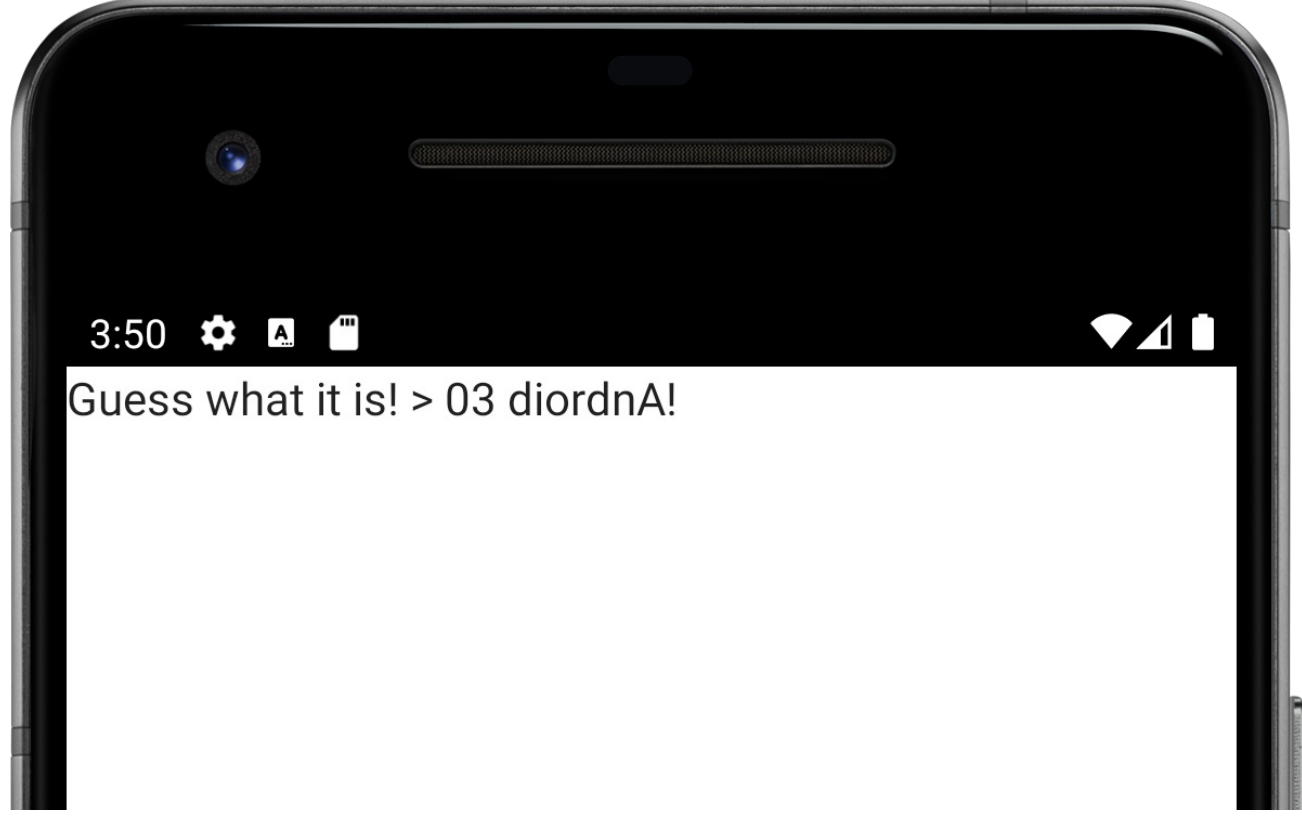


Common Kotlin file

2. Update the shared code by using reversed(), the Kotlin standard library function for reversing text that works on all platforms:

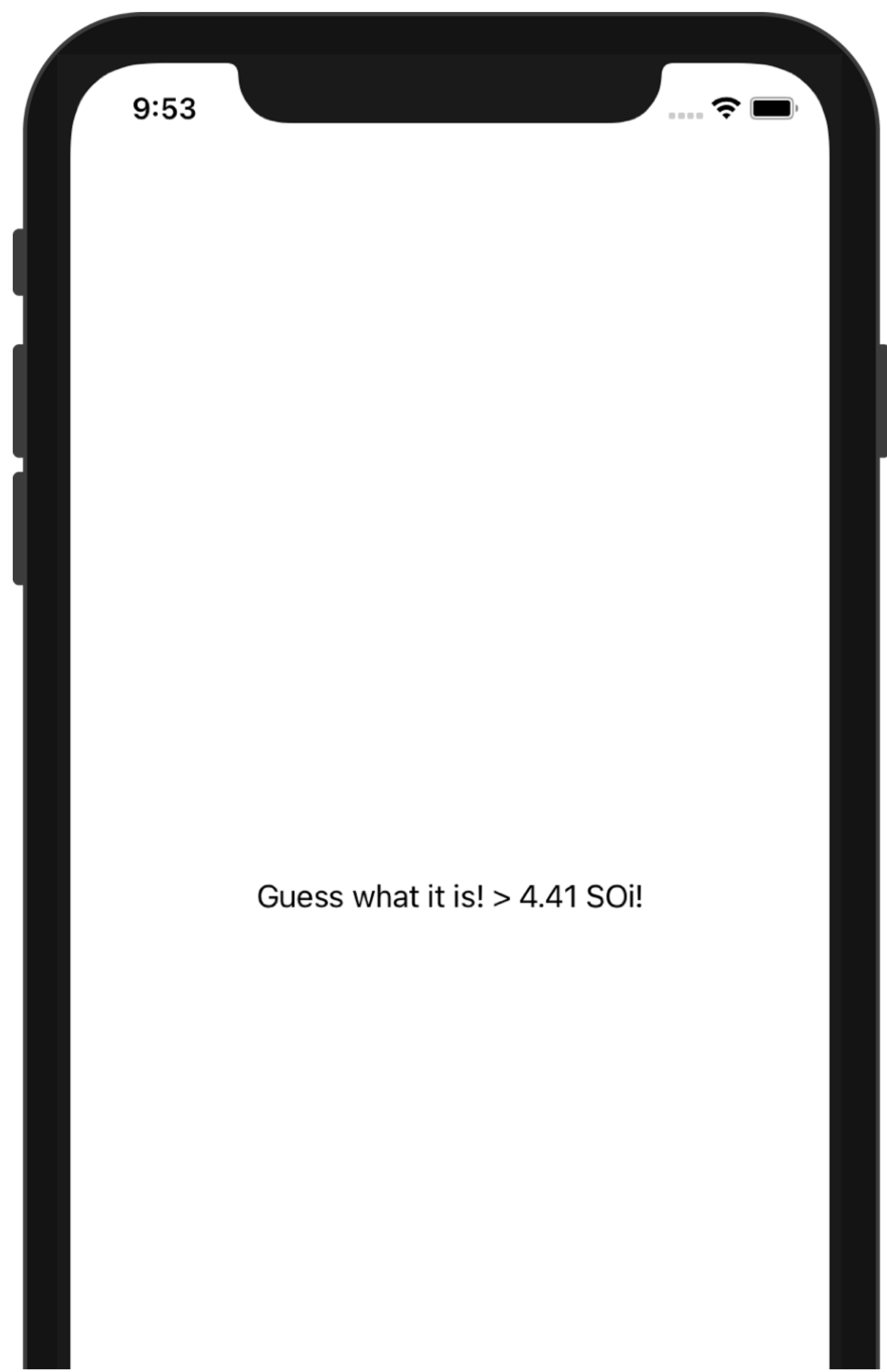
```kotlin
class Greeting {
    private val platform: Platform = getPlatform()

    fun greet(): String {
        return "Guess what it is! > ${platform.name.reversed()}!"
    }
}
```

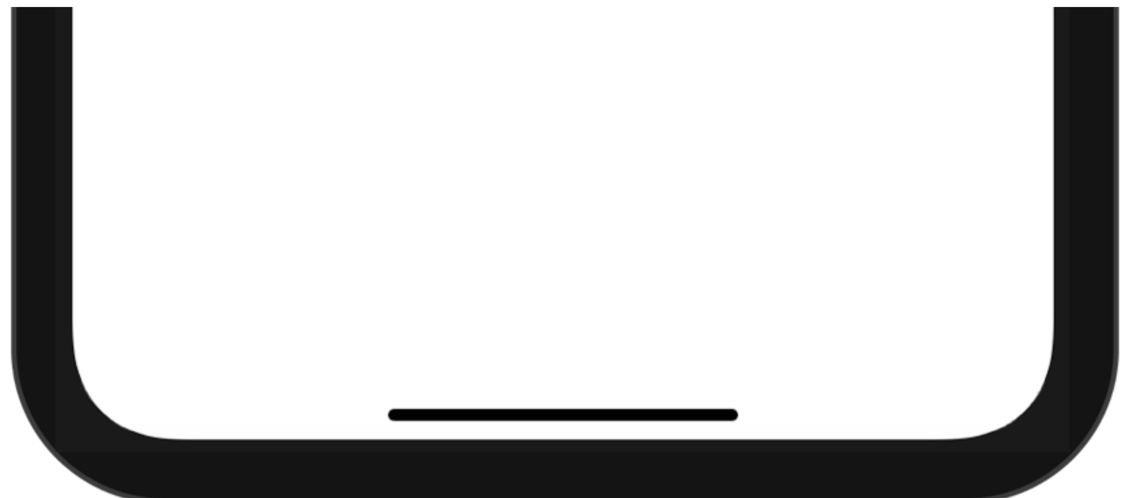
3. Re-run the androidApp configuration to see the updated application in the Android simulated device.



Updated mobile multiplatform app on Android

4. In Android Studio, switch to iosApp and re-run it to see the updated application in the iOS simulated device.





Updated mobile multiplatform app on iOS

Next step

In the next part of the tutorial, you'll learn about dependencies and add a third-party library to expand the functionality of your project.

Proceed to the next part

See also

- See how to create and run multiplatform tests to check that the code works correctly.
- Learn more about the project structure, the shared module's artifacts, and how the Android and iOS apps are produced.

Get help

- Kotlin Slack. Get an invite and join the #multiplatform channel.
- Kotlin issue tracker. Report a new issue.

Add dependencies to your project

You've already created your first cross-platform Kotlin Multiplatform project! Now let's learn how to add dependencies to third-party libraries, which is necessary for building successful cross-platform applications.

Dependency types

There are two types of dependencies that you can use in Multiplatform Mobile projects:

- Multiplatform dependencies. These are multiplatform libraries that support multiple targets and can be used in the common source set, commonMain.

  Many modern Android libraries already have multiplatform support, like Koin, Apollo, and Okio.

- Native dependencies. These are regular libraries from relevant ecosystems. You usually work with them in native iOS projects using CocoaPods or another dependency manager and in Android projects using Gradle.

When you work with a shared module, you can also depend on native dependencies and use them in the native source sets, androidMain and iosMain. Typically, you'll need these dependencies when you want to work with platform APIs, for example, security storage, and there is common logic.

For both types of dependencies, you can use local and external repositories.

Add a multiplatform dependency

> If you have experience developing Android apps, adding a multiplatform dependency is similar to adding a Gradle dependency in a regular Android project. The only difference is that you need to specify the source set.

Let's now go back to the app and make the greeting a little more festive. In addition to the device information, add a function to display the number of days left until New Year's Day. The kotlinx-datetime library, which has full multiplatform support, is the most convenient way to work with dates in your shared code.

1. Navigate to the build.gradle.kts file in the shared directory.
2. Add the following dependency to the commonMain source set dependencies:

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.4.0")
            }
        }
    }
}
```

3. Synchronize the Gradle files by clicking Sync Now in the notification.

Gradle files have changed since last project sync. A project syn... Sync Now Ignore these changes

Synchronize the Gradle files

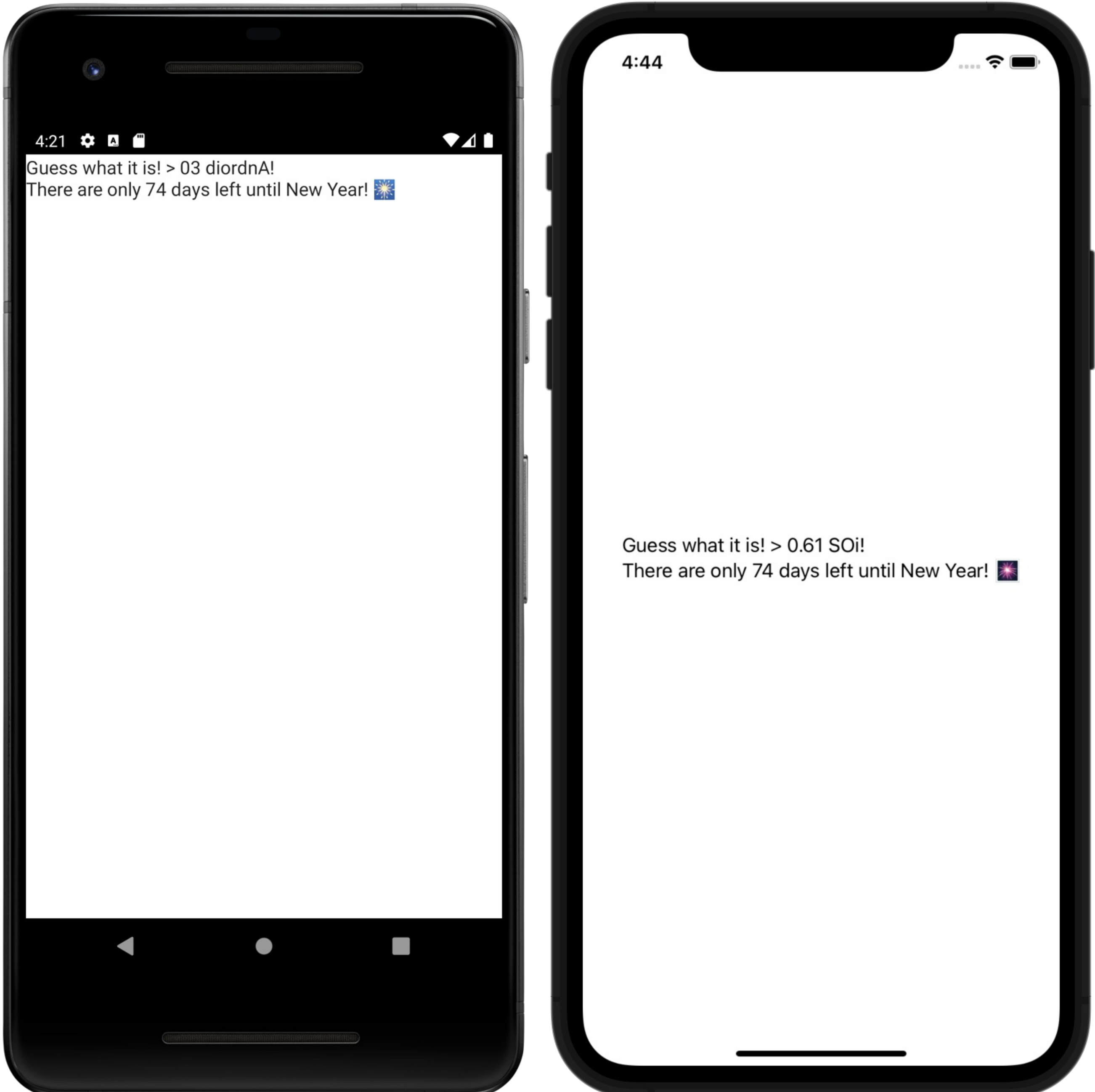
4. In shared/src/commonMain/kotlin, create a new file NewYear.kt in the project folder.
5. Update the file with a short function that calculates the number of days from today until the New Year using the date-time date arithmetic:

```
import kotlinx.datetime.*

fun daysUntilNewYear(): Int {
    val today = Clock.System.todayIn(TimeZone.currentSystemDefault())
    val closestNewYear = LocalDate(today.year + 1, 1, 1)
    return today.daysUntil(closestNewYear)
}
```

6. In Greeting.kt, update the greet() function to see the result:

```
class Greeting {
    private val platform: Platform = getPlatform()

    fun greet(): String {
        return "Guess what it is! > ${platform.name.reversed()}!" +
                "\nThere are only ${daysUntilNewYear()} days left until New Year! [illegible]"
    }
}
```

7. To see the results, re-run your androidApp and iosApp configurations from Android Studio:



Updated mobile multiplatform app with external dependencies

Next step

In the next part of the tutorial, you'll add more dependencies and more complex logic to your project.

Proceed to the next part

See also

- Discover how to work with multiplatform dependencies of all kinds: Kotlin libraries, Kotlin Multiplatform libraries, and other multiplatform projects.
- Learn how to add Android dependencies and iOS dependencies with or without CocoaPods for use in platform-specific source sets.
- Check out the examples of how to use Android and iOS libraries in sample projects (be sure to check the Platform APIs column).

Get help

- Kotlin Slack. Get an invite and join the #multiplatform channel.
- Kotlin issue tracker. Report a new issue.

Upgrade your app

You've already implemented common logic using external dependencies. Now you can add more complex logic. Network requests and data serialization are the most popular cases to share with Kotlin Multiplatform. Learn how to implement these in your first application, so that after completing this onboarding journey you can use them in future projects.

The updated app will retrieve data over the internet from a SpaceX API and display the date of the last successful launch of a SpaceX rocket.

Add more dependencies

You'll need the following multiplatform libraries in your project:

- kotlinx.coroutines, for using coroutines to write asynchronous code, which allows simultaneous operations.
- kotlinx.serialization, for deserializing JSON responses into objects of entity classes used to process network operations.
- Ktor, a framework as an HTTP client for retrieving data over the internet.

kotlinx.coroutines

To add kotlinx.coroutines to your project, specify a dependency in the common source set. To do so, add the following line to the build.gradle.kts file of the shared module:

```
sourceSets {
    val commonMain by getting {
        dependencies {
            // ...
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
        }
    }
}
```

The Multiplatform Gradle plugin automatically adds a dependency to the platform-specific (iOS and Android) parts of kotlinx.coroutines.

If you use Kotlin prior to version 1.7.20

If you use Kotlin 1.7.20 and later, you already have the new Kotlin/Native memory manager enabled by default. If it's not the case, add the following to the end of the build.gradle.kts file:

```
kotlin.targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget::class.java) {
    binaries.all {
        binaryOptions["memoryModel"] = "experimental"
    }
}
```

kotlinx.serialization

For kotlinx.serialization, you need the plugin required by the build system. The Kotlin serialization plugin is shipped with the Kotlin compiler distribution, and the IntelliJ IDEA plugin is bundled into the Kotlin plugin.

You can set up the serialization plugin with the Kotlin plugin using the Gradle plugins DSL by adding this line to the existing plugins block at the very beginning of the build.gradle.kts file in the shared module:

```
plugins {
    // ...
    kotlin("plugin.serialization") version "1.9.0"
}
```

Ktor

You can add Ktor in the same way you've added the kotlinx.coroutines library. In addition to specifying the core dependency (ktor-client-core) in the common source set, you also need to:

- Add the ContentNegotiation functionality (ktor-client-content-negotiation), responsible for serializing/deserializing the content in a specific format.
- Add the ktor-serialization-kotlinx-json dependency to instruct Ktor to use the JSON format and kotlinx.serialization as a serialization library. Ktor will expect JSON data and deserialize it into a data class when receiving responses.
- Provide the platform engines by adding dependencies on the corresponding artifacts in the platform source sets (ktor-client-android, ktor-client-darwin).

```
val ktorVersion = "2.3.2"

sourceSets {
    val commonMain by getting {
        dependencies {
            // ...
            implementation("io.ktor:ktor-client-core:$ktorVersion")
            implementation("io.ktor:ktor-client-content-negotiation:$ktorVersion")
            implementation("io.ktor:ktor-serialization-kotlinx-json:$ktorVersion")
        }
    }
    val androidMain by getting {
        dependencies {
            implementation("io.ktor:ktor-client-android:$ktorVersion")
        }
    }
    val iosMain by creating {
        // ...
        dependencies {
            implementation("io.ktor:ktor-client-darwin:$ktorVersion")
        }
    }
}
```

Synchronize the Gradle files by clicking Sync Now in the notification.

Create API requests

You'll need the SpaceX API to retrieve data and a single method to get the list of all launches from the v4/launches endpoint.

Add data model

In shared/src/commonMain/kotlin, create a new RocketLaunch.kt file in the project folder and add a data class which stores data from the SpaceX API:

```
import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable

@Serializable
data class RocketLaunch (
    @SerialName("flight_number")
    val flightNumber: Int,
    @SerialName("name")
    val missionName: String,
    @SerialName("date_utc")
    val launchDateUTC: String,
    @SerialName("success")
    val launchSuccess: Boolean?,
)
```

- The RocketLaunch class is marked with the @Serializable annotation, so that the kotlinx.serialization plugin can automatically generate a default serializer for it.
- The @SerialName annotation allows you to redefine field names, making it possible to declare properties in data classes with more readable names.

Connect HTTP client

1. In Greeting.kt, create a Ktor HTTPClient instance to execute network requests and parse the resulting JSON:

```
import io.ktor.client.*
import io.ktor.client.plugins.contentnegotiation.*
```

```
import io.ktor.serialization.kotlinx.json.*
import kotlinx.serialization.json.Json

class Greeting {
    private val platform: Platform = getPlatform()

    private val httpClient = HttpClient {
        install(ContentNegotiation) {
            json(Json {
                prettyPrint = true
                isLenient = true
                ignoreUnknownKeys = true
            })
        }
    }
}
```

To deserialize the result of the GET request, the ContentNegotiation Ktor plugin and the JSON serializer are used.

2. In the greet() function, retrieve the information about rocket launches by calling the httpClient.get() method and find the latest launch:

```
import io.ktor.client.call.*
import io.ktor.client.request.*

class Greeting {
    // ...
    @Throws(Exception::class)
    suspend fun greet(): String {
        val rockets: List<RocketLaunch> =
            httpClient.get("https://api.spacexdata.com/v4/launches").body()
        val lastSuccessLaunch = rockets.last { it.launchSuccess == true }
        return "Guess what it is! > ${platform.name.reversed()}!" +
                "\nThere are only ${daysUntilNewYear()} left until New Year! □□" +
                "\nThe last successful launch was ${lastSuccessLaunch.launchDateUTC} □□"
    }
}
```

The suspend modifier in the greet() function is necessary because it now contains a call to get(). It's a suspend function that has an asynchronous operation to retrieve data over the internet and can only be called from within a coroutine or another suspend function. The network request will be executed in the HTTP client's thread pool.

Add internet access permission

To access the internet, the Android application needs appropriate permission. Since all network requests are made from the shared module, it makes sense to add the internet access permission to its manifest.

Update your androidApp/src/main/AndroidManifest.xml file with the access permission:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.jetbrains.simplelogin.kotlinmultiplatformsandbox" >
    <uses-permission android:name="android.permission.INTERNET"/>
</manifest>
```

Update Android and iOS apps

You've already updated the API of the shared module by adding the suspend modifier to the greet() function. Now you need to update native (iOS, Android) parts of the project, so they can properly handle the result of calling the greet() function.

Android app

As both the shared module and the Android application are written in Kotlin, using shared code from Android is straightforward:

In androidApp/src/main/java, locate the MainActivity.kt file and update the following class replacing previous implementation:

```
import androidx.compose.runtime.*

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
        setContent {
            MyApplicationTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colors.background
                ) {
                    var text by remember { mutableStateOf("Loading") }
                    LaunchedEffect(true) {
                        text = try {
                            Greeting().greet()
                        } catch (e: Exception) {
                            e.localizedMessage ?: "error"
                        }
                    }
                    GreetingView(text)
                }
            }
        }
    }
}
```

The greet() function is now called inside LaunchedEffect to avoid recalling it on each recomposition.

iOS app

For the iOS part of the project, you'll make use of SwiftUI to build the user interface and the Model–view–viewmodel pattern to connect the UI to the shared module, which contains all the business logic.

The module is already connected to the iOS project — the Android Studio plugin wizard did all the configuration. The module is already imported and used in ContentView.swift with import shared.

> If you see errors in Xcode regarding the shared module or when updating your code, run the iosApp from Android Studio.

1. Launch your Xcode app and select Open a project or file.
2. Navigate to your project, for example KotlinMultiplatformSandbox, and select the iosApp folder. Click Open.
3. In iosApp/iOSApp.swift, update the entry point for your app:

```
@main
struct iOSApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView(viewModel: ContentView.ViewModel())
        }
    }
}
```

4. In iosApp/ContentView.swift, create a ViewModel class for ContentView, which will prepare and manage data for it:

```
import SwiftUI
import shared

struct ContentView: View {
    @ObservedObject private(set) var viewModel: ViewModel

    var body: some View {
        Text(viewModel.text)
    }
}

extension ContentView {
    class ViewModel: ObservableObject {
        @Published var text = "Loading..."
        init() {
            // Data will be loaded here
        }
    }
}
```

- ViewModel is declared as an extension to ContentView, as they are closely connected.
- The Combine framework connects the view model (ContentView.ViewModel) with the view (ContentView).
- ContentView.ViewModel is declared as an ObservableObject.
- The @Published wrapper is used for the text property.
- The @ObservedObject property wrapper is used to subscribe to the view model.

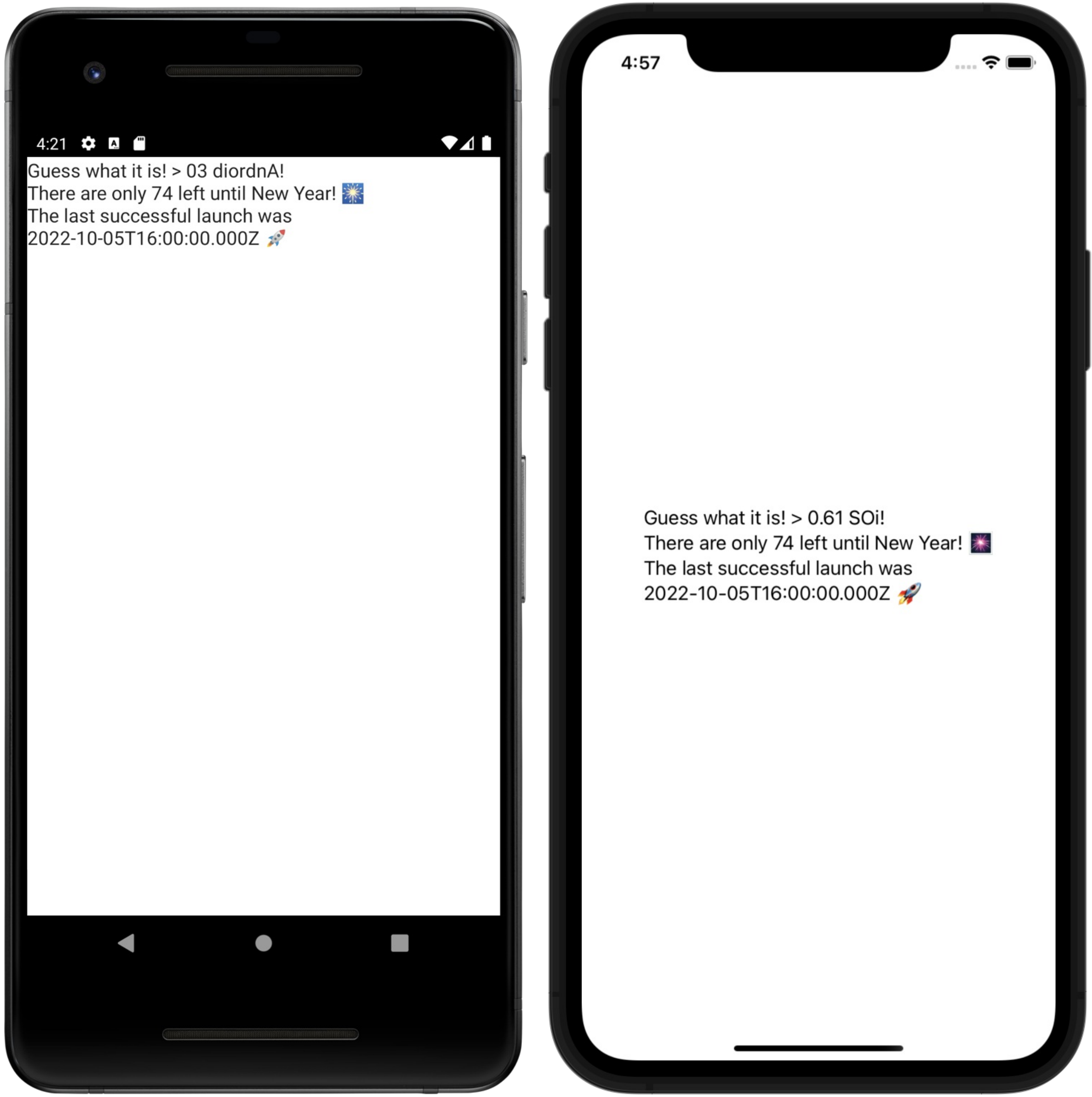
Now the view model will emit signals whenever this property changes.

5. Call the greet() function, which now also loads data from the SpaceX API, and save the result in the text property:

```
class ViewModel: ObservableObject {
    @Published var text = "Loading..."
    init() {
        Greeting().greet { greeting, error in
            DispatchQueue.main.async {
                if let greeting = greeting {
                    self.text = greeting
                } else {
                    self.text = error?.localizedDescription ?? "error"
                }
            }
        }
    }
}
```

- Kotlin/Native provides bidirectional interoperability with Objective-C, thus Kotlin concepts, including suspend functions, are mapped to the corresponding Swift/Objective-C concepts and vice versa. When you compile a Kotlin module into an Apple framework, suspending functions are available in it as functions with callbacks (completionHandler).
- The greet() function was marked with the @Throws(Exception::class) annotation. So any exceptions that are instances of the Exception class or its subclass will be propagated as NSError, so you can handle them in the completionHandler.
- When calling Kotlin suspend functions from Swift, completion handlers might be called on threads other than main, see the iOS integration in the Kotlin/Native memory manager. That's why DispatchQueue.main.async is used to update text property.

6. Re-run both androidApp and iosApp configurations from Android Studio to make sure your app's logic is synced:



Final results

Next step

In the final part of the tutorial, you'll wrap up your project and see what steps to take next.

Proceed to the next part

See also

- Explore various approaches to composition of suspending functions.
- Learn more about the interoperability with Objective-C frameworks and libraries.
- Complete this tutorial on networking and data storage.

Get help

- Kotlin Slack. Get an invite and join the #multiplatform channel.
- Kotlin issue tracker. Report a new issue.

Wrap up your project

You've created your first Multiplatform Mobile app that works both on iOS and Android! Now you know how to set up an environment for cross-platform mobile development, create a project in Android Studio, run your app on devices, and expand its functionality.

Now that you've gained some experience with Kotlin Multiplatform, you can take a look at some advanced topics and take on additional cross-platform mobile development tasks:

Next steps	Deep dive
• Add tests to your Kotlin Multiplatform project • Publish your mobile application to app stores • Introduce cross-platform mobile development to your team • Check out the list of useful tools and libraries	• Kotlin Multiplatform for mobile project structure • Interoperability with Objective-C frameworks and libraries • Adding dependencies on multiplatform libraries • Adding Android dependencies • Adding iOS dependencies

Tutorials and samples	Community and feedback
• Make your Android app cross-platform • Create a multiplatform app using Ktor and SQLDelight • Share UIs between iOS and Android using Compose Multiplatform • See the curated list of sample projects	• Join the #multiplatform channel in Kotlin Slack • Subscribe to the "kotlin-multiplatform" tag on Stack Overflow • Subscribe to the Kotlin YouTube channel • Report a problem to our issue tracker

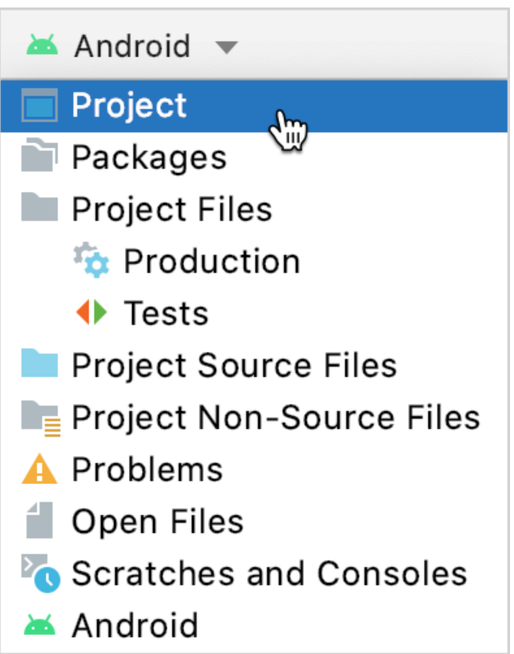
Understand mobile project structure

The purpose of the Kotlin Multiplatform technology is unifying the development of applications with common logic for Android and iOS platforms. To make this possible, it uses a mobile-specific structure of Kotlin Multiplatform projects.

This page describes the structure and components of a basic cross-platform mobile project: shared module, Android app, and an iOS app.

> This structure isn't the only possible way to organize your project; however, we recommend it as a starting point.

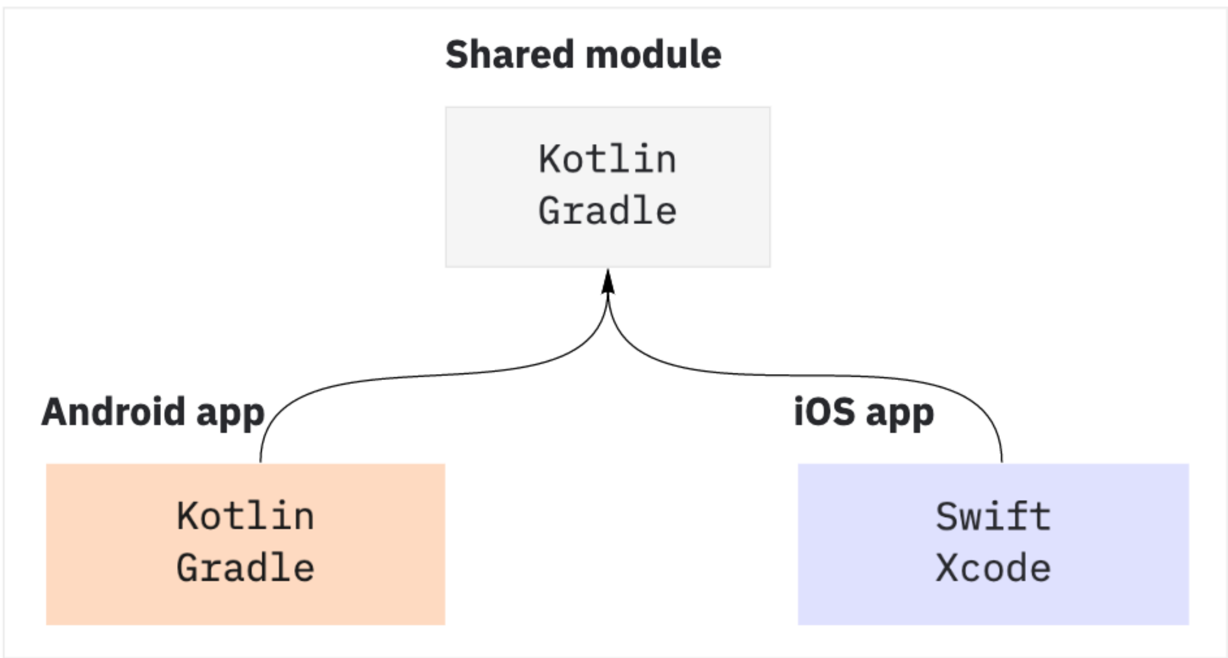
To view the complete structure of your mobile multiplatform project, switch the view from Android to Project.



Select the Project view

Root project

The root project is a Gradle project that holds the shared module and the Android application as its subprojects. They are linked together via the Gradle multi-project mechanism.



Basic Multiplatform Mobile project structure

Kotlin

```
// settings.gradle.kts
```

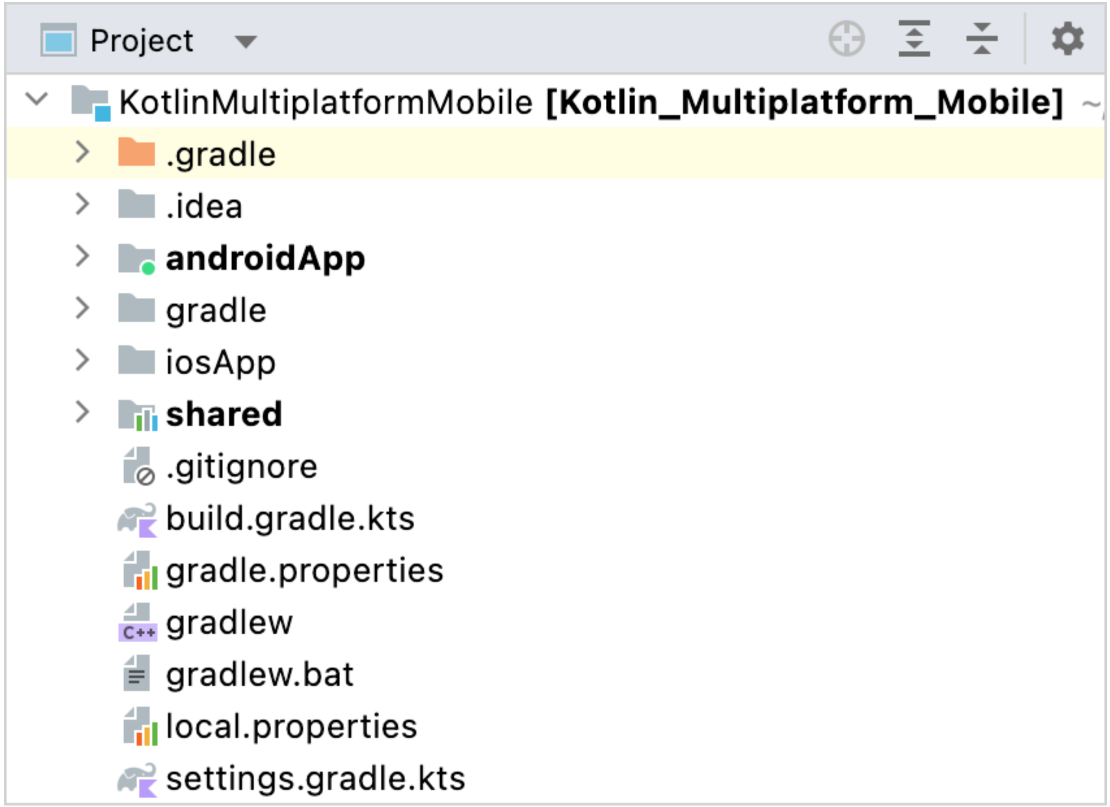
```
include(":shared")
include(":androidApp")
```

Groovy

```
// settings.gradle
include ':shared'
include ':androidApp'
```

The iOS application is produced from an Xcode project. It's stored in a separate directory within the root project. Xcode uses its own build system; thus, the iOS application project isn't connected with other parts of the Multiplatform Mobile project via Gradle. Instead, it uses the shared module as an external artifact – framework. For details on integration between the shared module and the iOS application, see iOS application.

This is a basic structure of a cross-platform mobile project:



Basic Multiplatform Mobile project directories

The root project does not hold source code. You can use it to store global configuration in its build.gradle(.kts) or gradle.properties, for example, add repositories or define global configuration variables.

For more complex projects, you can add more modules into the root project by creating them in the IDE and linking via include declarations in the Gradle settings.

Shared module

Shared module contains the core application logic used in both Android and iOS target platforms: classes, functions, and so on. This is a Kotlin Multiplatform module that compiles into an Android library and an iOS framework. It uses the Gradle build system with the Kotlin Multiplatform plugin applied and has targets for Android and iOS.

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
    // ..
```

```
}

kotlin {
    android()
    ios()
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
    //..
}

kotlin {
    android()
    ios()
}
```

Source sets

The shared module contains the code that is common for Android and iOS applications. However, to implement the same logic on Android and iOS, you sometimes need to write two platform-specific versions of it. To handle such cases, Kotlin offers the expect/actual mechanism. The source code of the shared module is organized in three source sets accordingly:

- commonMain stores the code that works on both platforms, including the expect declarations
- androidMain stores Android-specific parts, including actual implementations
- iosMain stores iOS-specific parts, including actual implementations

Each source set has its own dependencies. Kotlin standard library is added automatically to all source sets, you don't need to declare it in the build script.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting
        val androidMain by getting {
            dependencies {
                implementation("androidx.core:core-ktx:1.2.0")
            }
        }
        val iosMain by getting
        // ...
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
        }
        androidMain {
            dependencies {
                implementation 'androidx.core:core-ktx:1.2.0'
            }
        }
        iosMain {
        }

        // ...
    }
}
```

When you write your code, add the dependencies you need to the corresponding source sets. Read Multiplatform documentation on adding dependencies for more information.

Along with *Main source sets, there are three matching test source sets:

- commonTest
- androidTest
- iosTest

Use them to store unit tests for common and platform-specific source sets accordingly. By default, they have dependencies on Kotlin test library, providing you with means for Kotlin unit testing: annotations, assertion functions and other. You can add dependencies on other test libraries you need.

Kotlin

```
kotlin {
    sourceSets {
        // ...
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
        val androidTest by getting
        val iosTest by getting
    }

}
```

Groovy

```
kotlin {
    sourceSets {
        //...

        commonTest {
            dependencies {
                implementation kotlin('test')
            }
        }
        androidTest {

        }
        iosTest {

        }
    }
}
```

The main and test source sets described above are default. The Kotlin Multiplatform plugin generates them automatically upon target creation. In your project, you can add more source sets for specific purposes. For more information, see Multiplatform DSL reference.

Android library

The configuration of the Android library produced from the shared module is typical for Android projects. To learn about Android libraries creation, see Create an Android library in the Android developer documentation.

To produce the Android library, a separate Gradle plugin is used in addition to Kotlin Multiplatform:

Kotlin

```
plugins {
    // ...
    id("com.android.library")
}
```

Groovy

```
plugins {
    // ...
    id 'com.android.library'
}
```

The configuration of Android library is stored in the android {} top-level block of the shared module's build script:

Kotlin

```
android {
    compileSdk = 29
    sourceSets["main"].manifest.srcFile("src/androidMain/AndroidManifest.xml")
    defaultConfig {
        minSdk = 24
        targetSdk = 29
    }
}
```

Groovy

```
android {
    compileSdk 29
    sourceSets.main.manifest.srcFile 'src/androidMain/AndroidManifest.xml'
    defaultConfig {
        minSdk 24
        targetSdk 29
    }
}
```

It's typical for any Android project. You can edit it to suit your needs. To learn more, see the Android developer documentation.

iOS framework

For using in iOS applications, the shared module compiles into a framework – a kind of hierarchical directory with shared resources used on the Apple platforms. This framework connects to the Xcode project that builds into an iOS application.

The framework is produced via the Kotlin/Native compiler. The framework configuration is stored in the ios {} block of the build script within kotlin {}. It defines the output type framework and the string identifier baseName that is used to form the name of the output artifact. Its default value matches the Gradle module name. For a real project, it's likely that you'll need a more complex configuration of the framework production. For details, see Multiplatform documentation.

Kotlin

```
kotlin {
    // ...
    ios {
        binaries {
            framework {
                baseName = "shared"
            }
        }
    }
}
```

Groovy

```
kotlin {
    // ...
    ios {
        binaries {
            framework {
                baseName = 'shared'
            }
        }
    }
}
```

Additionally, there is a Gradle task embedAndSignAppleFrameworkForXcode, that exposes the framework to the Xcode project the iOS application is built from. It uses the iOS application's project configuration to define the build mode (debug or release) and provide the appropriate framework version to the specified location.

The task is built into the multiplatform plugin. It executes upon each build of the Xcode project to provide the latest version of the framework for the iOS application. For details, see iOS application.

Use the embedAndSignAppleFrameworkForXcode Gradle task with Xcode project builds only; otherwise, you'll get an error.

Android application

The Android application part of a Multiplatform Mobile project is a typical Android application written in Kotlin. In a basic cross-platform mobile project, it uses two Gradle plugins:

- Kotlin Android
- Android Application

Kotlin

```
plugins {
    id("com.android.application")
    kotlin("android")
}
```

Groovy

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}
```

To access the shared module code, the Android application uses it as a project dependency.

Kotlin

```
dependencies {
    implementation(project(":shared"))
    //..
}
```

Groovy

```
dependencies {
    implementation project(':shared')
    //..
}
```

Besides this dependency, the Android application uses the Kotlin standard library (which is added automatically) and some common Android dependencies:

Kotlin

```
dependencies {
    //..
    implementation("androidx.core:core-ktx:1.2.0")
    implementation("androidx.appcompat:appcompat:1.1.0")
    implementation("androidx.constraintlayout:constraintlayout:1.1.3")
}
```

Groovy

```
dependencies {
    //..
    implementation 'androidx.core:core-ktx:1.2.0'
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
}
```

Add your project's Android-specific dependencies to this block. The build configuration of the Android application is located in the android {} top-level block of the build script:

Kotlin

```kotlin
android {
    compileSdk = 29
    defaultConfig {
        applicationId = "org.example.androidApp"
        minSdk = 24
        targetSdk = 29
        versionCode = 1
        versionName = "1.0"
    }
    buildTypes {
        getByName("release") {
            isMinifyEnabled = false
        }
    }
}
```

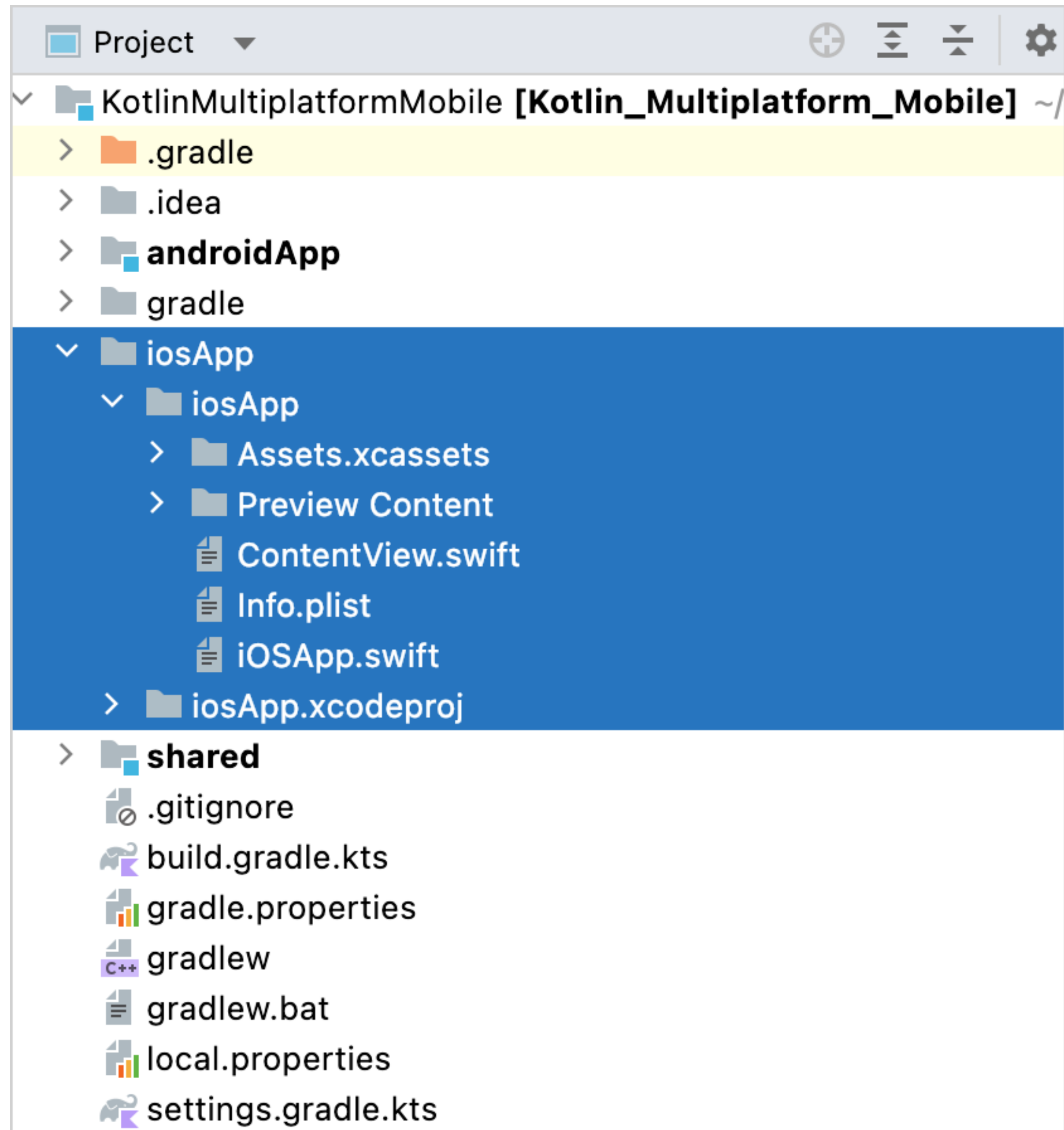
Groovy

```groovy
android {
    compileSdk 29
    defaultConfig {
        applicationId 'org.example.androidApp'
        minSdk 24
        targetSdk 29
        versionCode 1
        versionName '1.0'
    }
    buildTypes {
        'release' {
            minifyEnabled false
        }
    }
}
```

It's typical for any Android project. You can edit it to suit your needs. To learn more, see the Android developer documentation.

iOS application

The iOS application is produced from an Xcode project generated automatically by the New Project wizard. It resides in a separate directory within the root project.

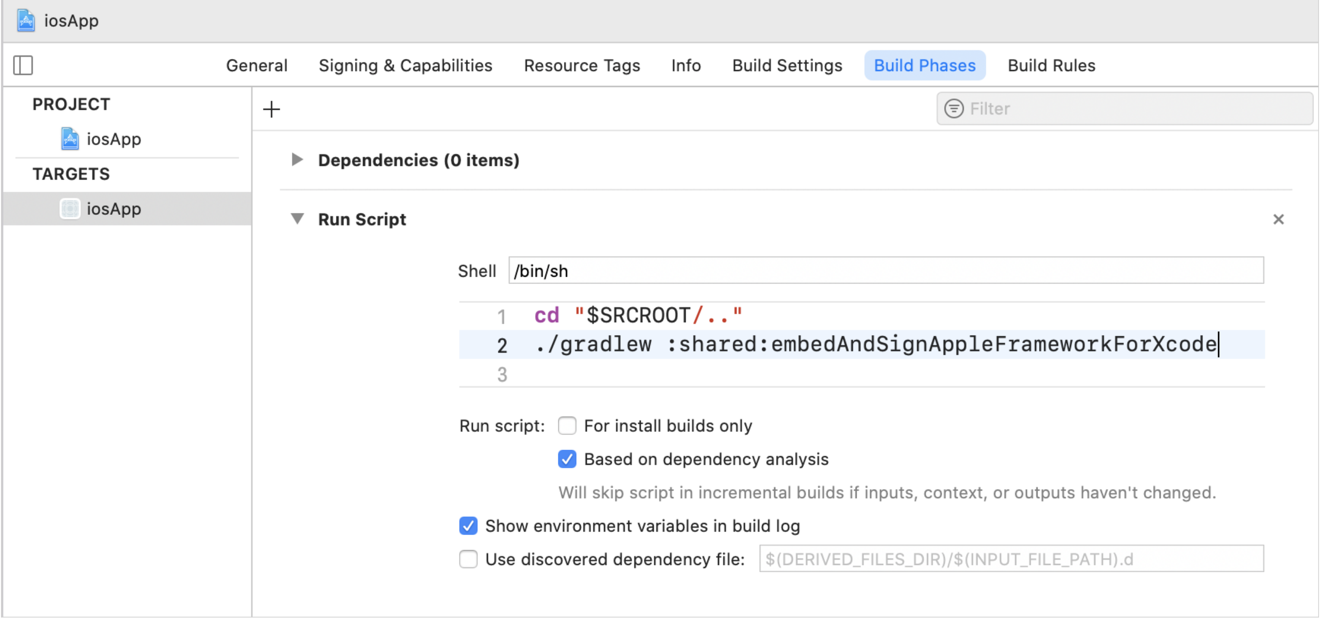


Basic Kotlin Multiplatform Xcode project

For each build of the iOS application, the project obtains the latest version of the framework. To do this, it uses a Run Script build phase that executes the embedAndSignAppleFrameworkForXcode Gradle task from the shared module. This task generates the .framework with the required configuration, depending on the Xcode environment settings, and puts the artifact into the DerivedData Xcode directory.

- If you have a custom name for the Apple framework, use embedAndSign<Custom-name>AppleFrameworkForXcode as the name for this Gradle task.
- If you have a custom build configuration that is different from the default Debug or Release, on the Build Settings tab, add the KOTLIN_FRAMEWORK_BUILD_TYPE setting under User-Defined and set it to Debug or Release.

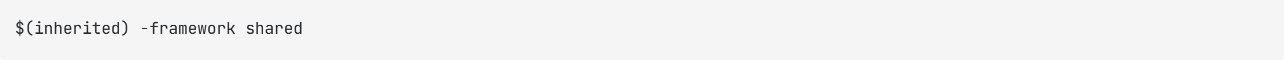
> Use the embedAndSignAppleFrameworkForXcode Gradle task with Xcode project builds only; otherwise, you'll get an error.



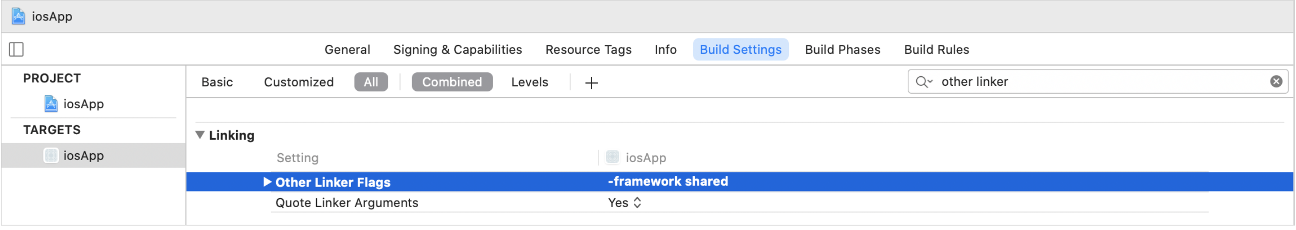
Execution of embedAndSignAppleFrameworkForXcode in the Xcode project settings

To embed framework into the application and make the declarations from the shared module available in the source code of the iOS application, the following build settings should be configured properly:

1. Other Linker flags under the Linking section:



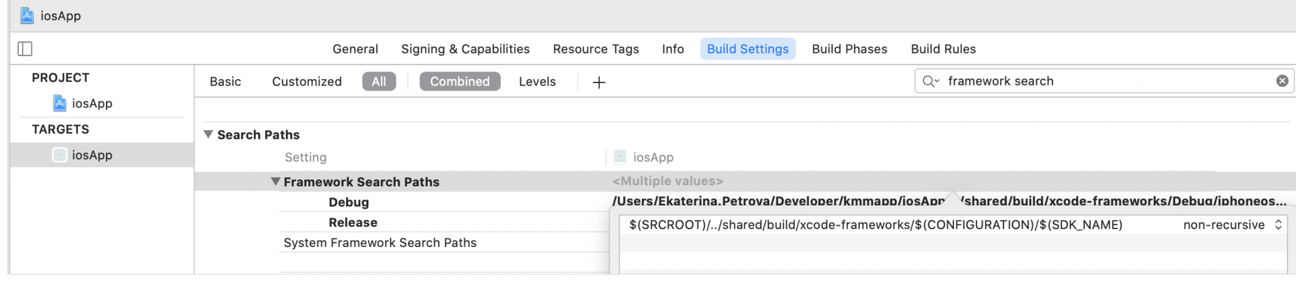
```
$(inherited) -framework shared
```



Configuring Other linker flags in the Xcode project settings

2. Framework Search Paths under the Search Paths section:

```
$(SRCROOT)/../shared/build/xcode-frameworks/$(CONFIGURATION)/$(SDK_NAME)
```



Configuring Framework Search Paths in the Xcode project settings

In other aspects, the Xcode part of a cross-platform mobile project is a typical iOS application project. To learn more about creating iOS application, see the Xcode documentation.

Make your Android application work on iOS – tutorial

Learn how to make your existing Android application cross-platform so that it works both on Android and iOS. You'll be able to write code and test it for both Android and iOS only once, in one place.

This tutorial uses a sample Android application with a single screen for entering a username and password. The credentials are validated and saved to an in-memory database.

> If you aren't familiar with Kotlin Multiplatform for mobile, learn how to set up environment and create a cross-platform application from scratch first.

Prepare an environment for development

1. Install all the necessary tools and update them to the latest versions.

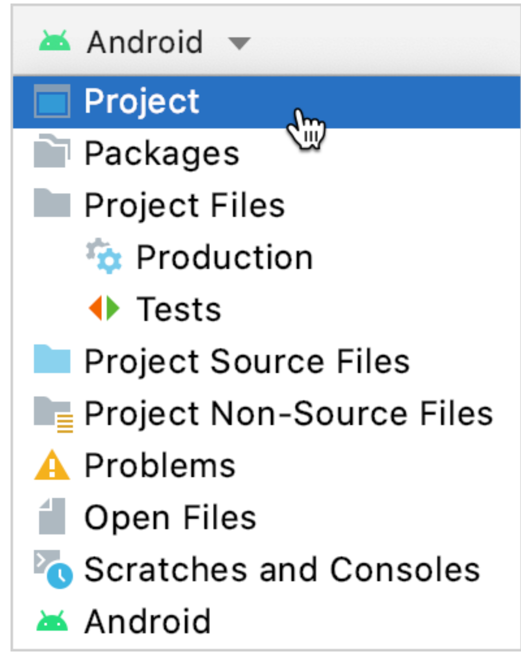
   > You will need a Mac with macOS to complete certain steps in this tutorial, which include writing iOS-specific code and running an iOS application. These steps can't be performed on other operating systems, such as Microsoft Windows. This is due to an Apple requirement.

2. In Android Studio, create a new project from version control:

```
https://github.com/Kotlin/kmm-integration-sample
```

   > The master branch contains the project's initial state — a simple Android application. To see the final state with the iOS application and the shared module, switch to the final branch.

3. Switch to the Project view.



Project view

Make your code cross-platform

To make your application work on iOS, you'll first make your code cross-platform, and then you'll reuse your cross-platform code in a new iOS application.

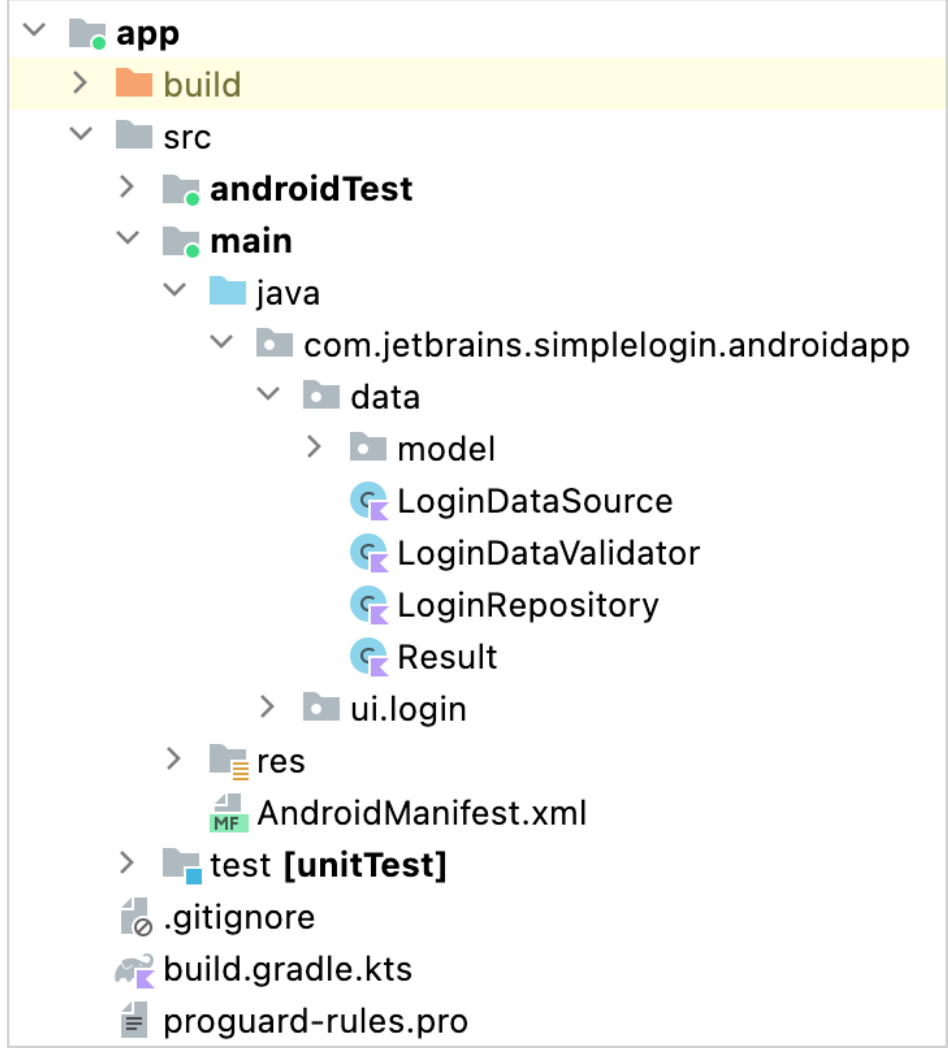
To make your code cross-platform:

1. Decide what code to make cross-platform.
2. Create a shared module for cross-platform code.
3. Add a dependency on the shared module to your Android application.
4. Make the business logic cross-platform.
5. Run your cross-platform application on Android.

Decide what code to make cross-platform

Decide which code of your Android application is better to share for iOS and which to keep native. A simple rule is: share what you want to reuse as much as possible. The business logic is often the same for both Android and iOS, so it's a great candidate for reuse.

In your sample Android application, the business logic is stored in the package com.jetbrains.simplelogin.androidapp.data. Your future iOS application will use the same logic, so you should make it cross-platform, as well.



Business logic to share

Create a shared module for cross-platform code

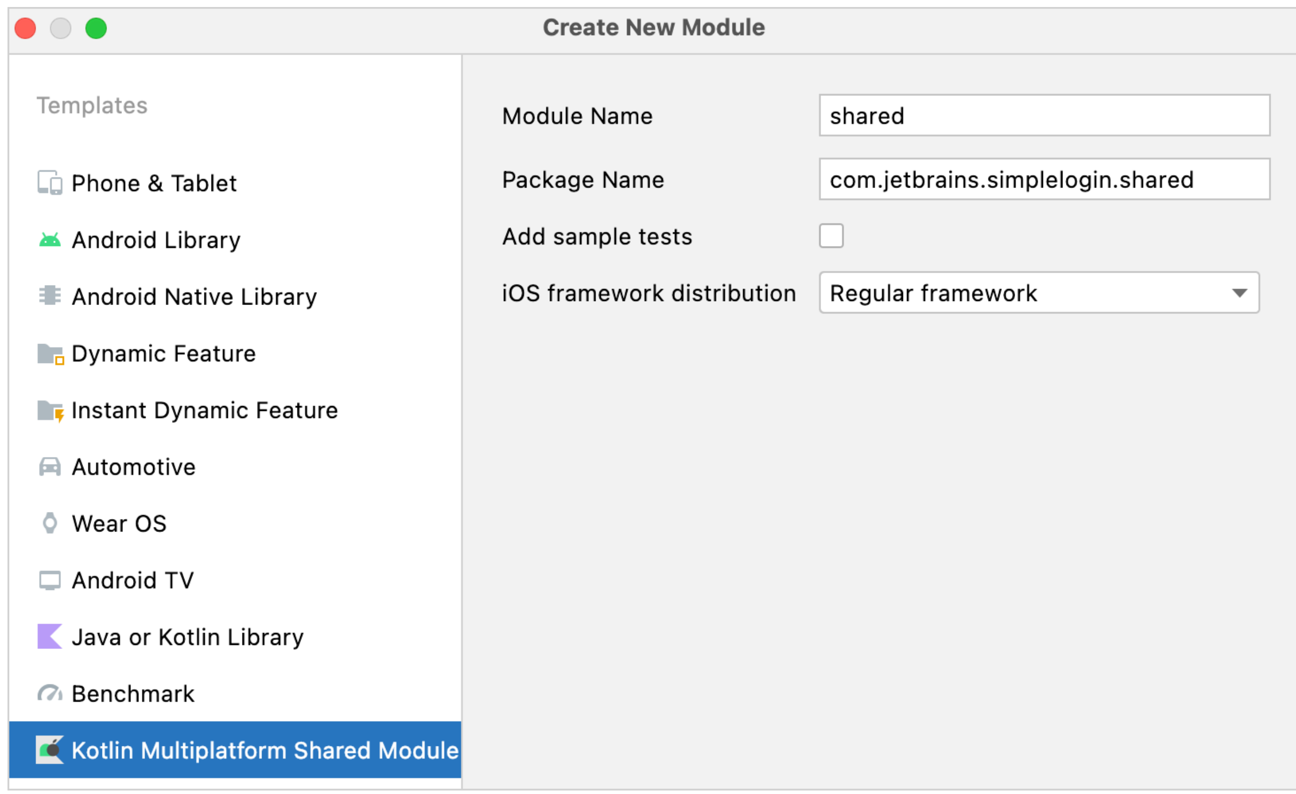
The cross-platform code that is used for both iOS and Android is stored in the shared module. The Kotlin Multiplatform plugin provides a special wizard for creating such modules.

In your Android project, create a Kotlin Multiplatform shared module for your cross-platform code. Later you'll connect it to your existing Android application and

your future iOS application.

1. In Android Studio, click File | New | New Module.

2. In the list of templates, select Kotlin Multiplatform Shared Module, enter the module name shared, and select the Regular framework in the list of iOS framework distribution options.
   This is required for connecting the shared module to the iOS application.



Kotlin Multiplatform shared module

3. Click Finish.

The wizard will create the Kotlin Multiplatform shared module, update the configuration files, and create files with classes that demonstrate the benefits of Kotlin Multiplatform. You can learn more about the project structure.

Add a dependency on the shared module to your Android application

To use cross-platform code in your Android application, connect the shared module to it, move the business logic code there, and make this code cross-platform.

1. In the build.gradle.kts file of the shared module, ensure that compileSdk and minSdk are the same as those in the build.gradle.kts of your Android application in the app module.

   If they're different, update them in the build.gradle.kts of the shared module. Otherwise, you'll encounter a compile error.

2. Add a dependency on the shared module to the build.gradle.kts of your Android application.

```
dependencies {
    implementation (project(":shared"))
}
```

3. Synchronize the Gradle files by clicking Sync Now in the notification.

Gradle files have changed since last project sync. A project syn... Sync Now Ignore these changes

Synchronize the Gradle files

4. In the app/src/main/java/ directory, open the LoginActivity class in the com.jetbrains.simplelogin.androidapp.ui.login package.
5. To make sure that the shared module is successfully connected to your application, dump the greet() function result to the log by updating the onCreate() method:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    Log.i("Login Activity", "Hello from shared module: " + (Greeting().greet()))
}
```

6. Follow Android Studio suggestions to import missing classes.
7. Debug the app. On the Logcat tab, search for Hello in the log, and you'll find the greeting from the shared module.

Logcat

Emulator Pixel_3a_API_30_x86 A | com.jetbrains.simplelogin.androida | Info | hello

logcat

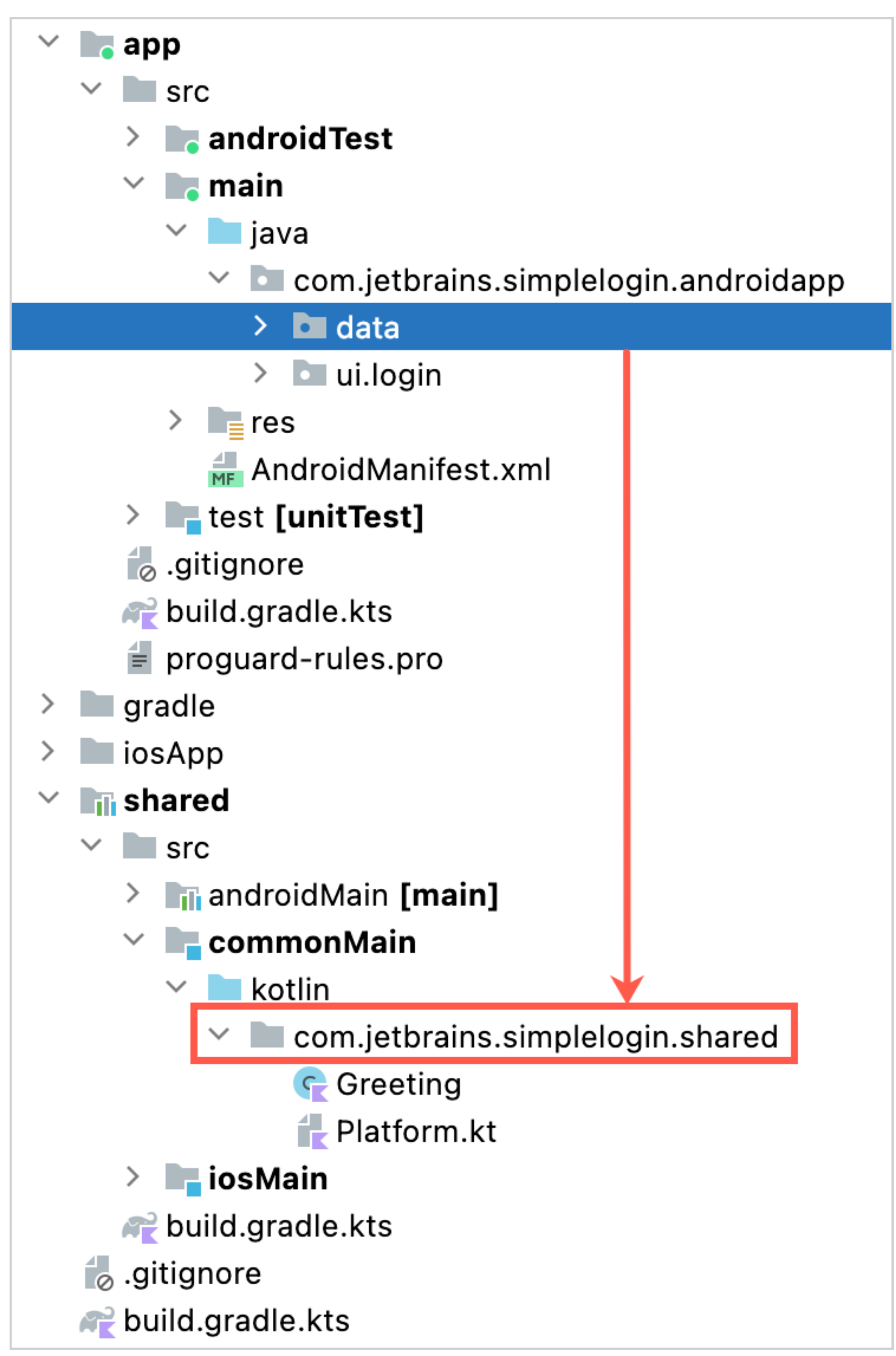
```
2021-01-27 16:56:05.008 7036-7036/com.jetbrains.simplelogin.androidapp I/Login Activity: Hello from shared module: Hello, Android 30!
```

Greeting from the shared module

Make the business logic cross-platform

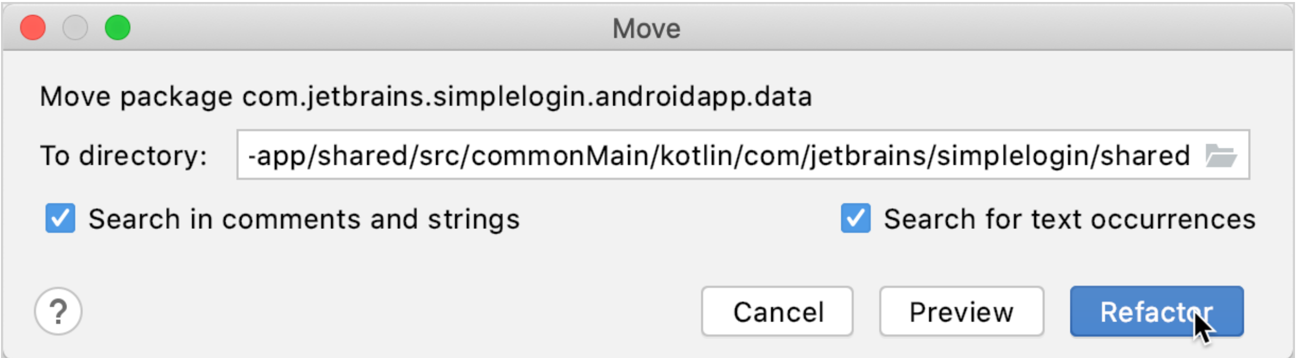
You can now extract the business logic code to the Kotlin Multiplatform shared module and make it platform-independent. This is necessary for reusing the code for both Android and iOS.

1. Move the business logic code com.jetbrains.simplelogin.androidapp.data from the app directory to the com.jetbrains.simplelogin.shared package in the shared/src/commonMain directory. You can drag and drop the package or refactor it by moving everything from one directory to another.



Drag and drop the package with the business logic code

2. When Android Studio asks what you'd like to do, select to move the package, and then approve the refactoring.



Refactor the business logic package

3. Ignore all warnings about platform-dependent code and click Continue.

Problems Detected

The following problems were found:

Field **Patterns.EMAIL_ADDRESS**, referenced in function **LoginDataValidator.isEmailValid(String)**, will not be accessible in modu **Simple_Login.shared.commonMain**

Class **java.util.UUID**, referenced in variable **com.jetbrains.simplelogin.androidapp.data.LoginDataSource.login.** be accessible in module **Simple_Login.shared.commonMain**

Class **android.util.Patterns**, referenced in function **LoginDataValidator.isEmailValid(String)**, will not be accessible in modu **Simple_Login.shared.commonMain**

Method **UUID.randomUUID()**, referenced in variable **com.jetbrains.simplelogin.androidapp.data.LoginDataSource.login.** be accessible in module **Simple_Login.shared.commonMain**

Constructor **IOException.IOException(String, Throwable)**, referenced ir **LoginDataSource.login(String, String)**, will not be accessible in module **Simple_Login.shared.commonMain**

Do you wish to ignore them and continue?

Show Conflicts in View | Cancel | Continue

Warnings about platform-dependent code

4. Remove Android-specific code by replacing it with cross-platform Kotlin code or connecting to Android-specific APIs using expect and actual declarations. See the following sections for details:

Replace Android-specific code with cross-platform code

To make your code work well on both Android and iOS, replace all JVM dependencies with Kotlin dependencies in the moved data directory wherever possible.

1. In the LoginDataSource class, replace IOException in the login() function with RuntimeException. IOException is not available in Kotlin.

```kotlin
// Before
return Result.Error(IOException("Error logging in", e))
```

```kotlin
// After
return Result.Error(RuntimeException("Error logging in", e))
```

2. In the LoginDataValidator class, replace the Patterns class from the android.utils package with a Kotlin regular expression matching the pattern for email validation:

```kotlin
// Before
private fun isEmailValid(email: String) = Patterns.EMAIL_ADDRESS.matcher(email).matches()
```

```kotlin
// After
private fun isEmailValid(email: String) = emailRegex.matches(email)

companion object {
    private val emailRegex =
        ("[a-zA-Z0-9\\+\\.\\_\\%\\-\\+]{1,256}" +
            "\\@" +
            "[a-zA-Z0-9][a-zA-Z0-9\\-]{0,64}" +
            "(" +
            "\\." +
            "[a-zA-Z0-9][a-zA-Z0-9\\-]{0,25}" +
            ")+").toRegex()
}
```

Connect to platform-specific APIs from the cross-platform code

In the LoginDataSource class, a universally unique identifier (UUID) for fakeUser is generated using the java.util.UUID class, which is not available for iOS.

```kotlin
val fakeUser = LoggedInUser(java.util.UUID.randomUUID().toString(), "Jane Doe")
```

Since the Kotlin standard library doesn't provide functionality for generating UUIDs, you still need to use platform-specific functionality for this case.

Provide the expect declaration for the randomUUID() function in the shared code and its actual implementations for each platform – Android and iOS – in the corresponding source sets. You can learn more about connecting to platform-specific APIs.

1. Remove the java.util.UUID class from the common code:

```kotlin
val fakeUser = LoggedInUser(randomUUID(), "Jane Doe")
```

2. Create the Utils.kt file in the com.jetbrains.simplelogin.shared package of the shared/src/commonMain directory and provide the expect declaration:

```kotlin
package com.jetbrains.simplelogin.shared

expect fun randomUUID(): String
```

3. Create the Utils.kt file in the com.jetbrains.simplelogin.shared package of the shared/src/androidMain directory and provide the actual implementation for randomUUID() in Android:

```kotlin
package com.jetbrains.simplelogin.shared

import java.util.*

actual fun randomUUID() = UUID.randomUUID().toString()
```

4. Create the Utils.kt file in the com.jetbrains.simplelogin.shared of the shared/src/iosMain directory and provide the actual implementation for randomUUID() in iOS:

```
package com.jetbrains.simplelogin.shared

import platform.Foundation.NSUUID

actual fun randomUUID(): String = NSUUID().UUIDString()
```

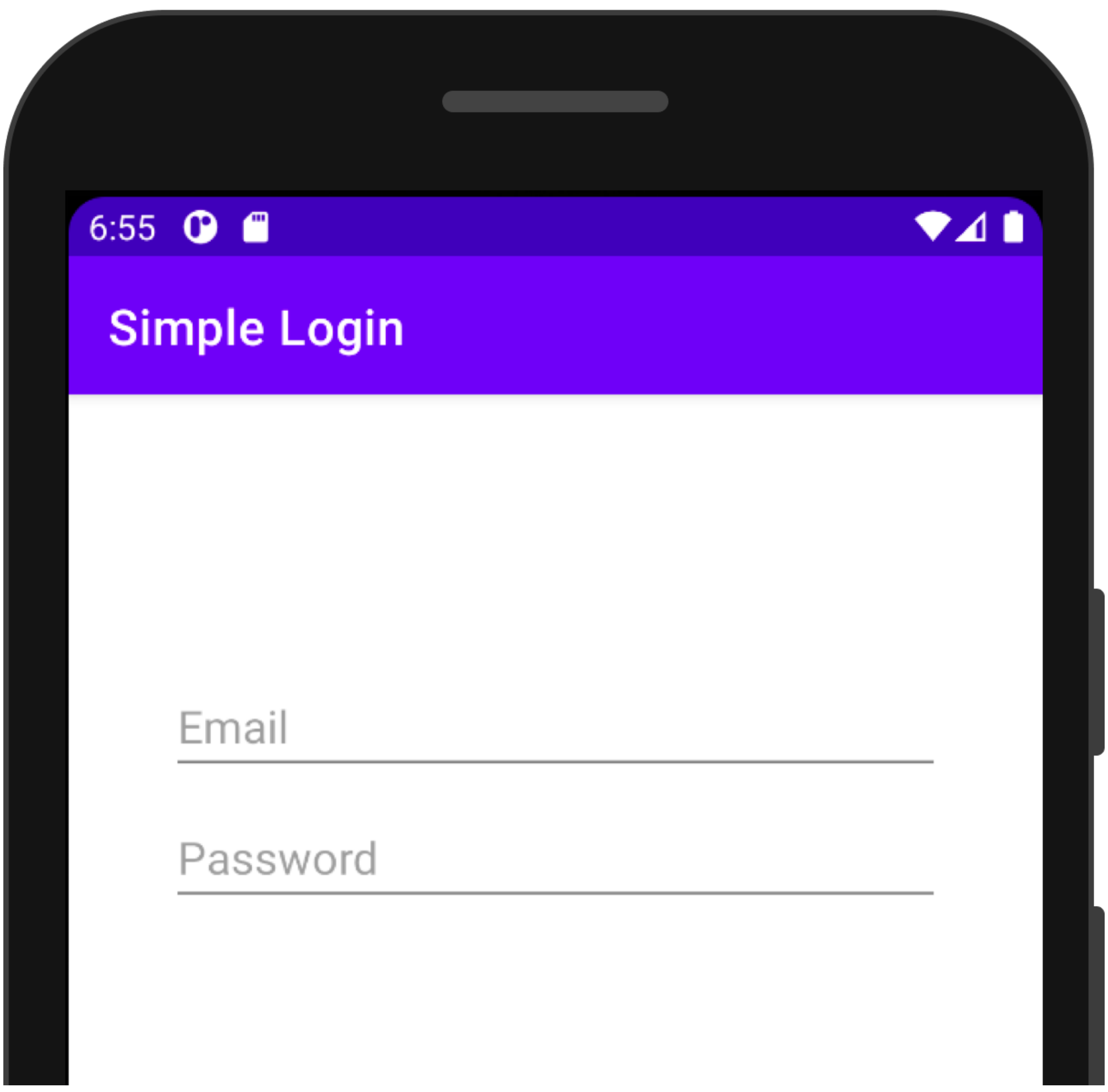
5. All it's left to do is to explicitly import randomUUID in the LoginDataSource.kt file of the shared/src/commonMain directory:

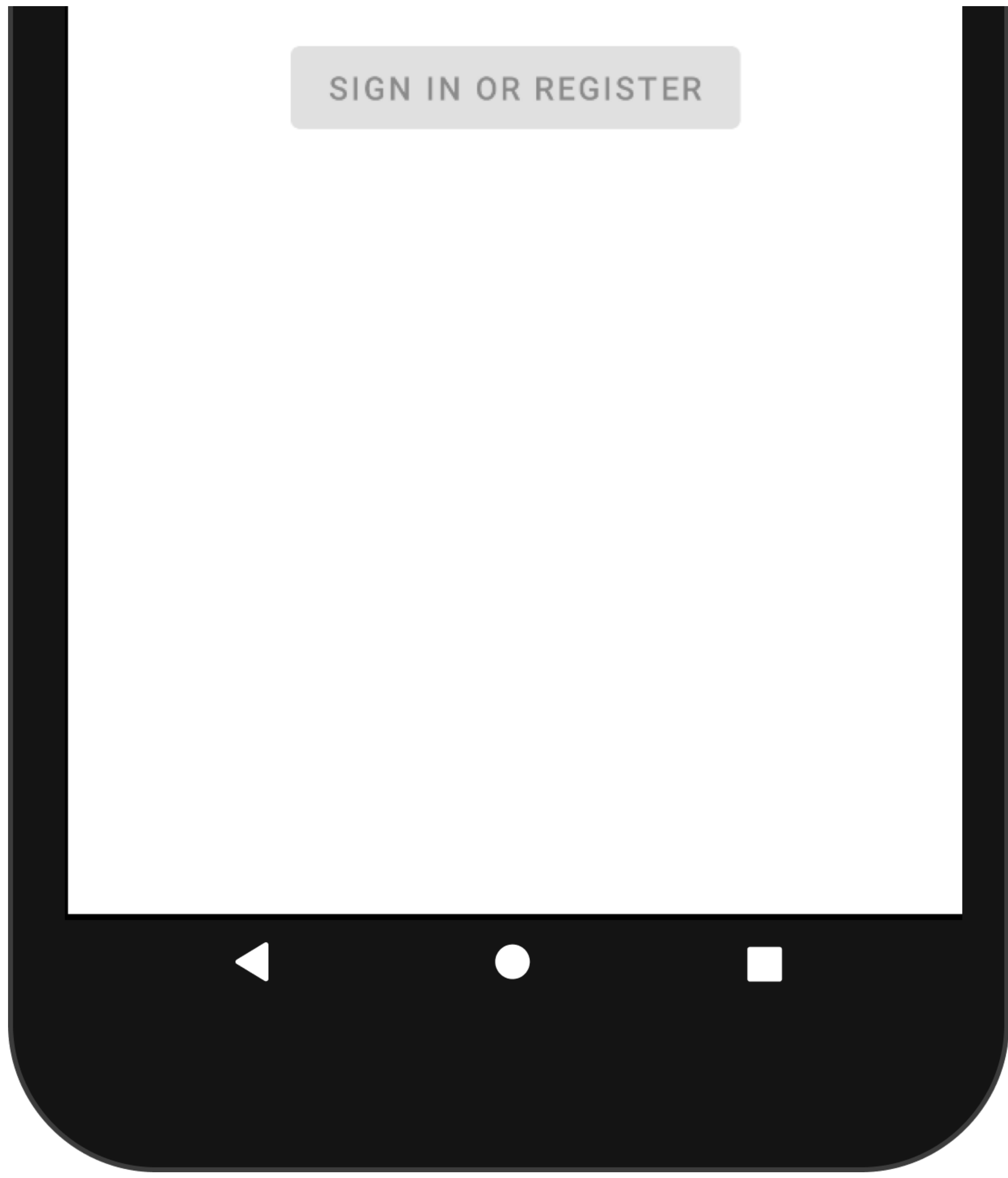
```
import com.jetbrains.simplelogin.shared.randomUUID
```

For Android and iOS, Kotlin will use its different platform-specific implementations.

Run your cross-platform application on Android

Run your cross-platform application for Android to make sure it works.





Android login application

Make your cross-platform application work on iOS

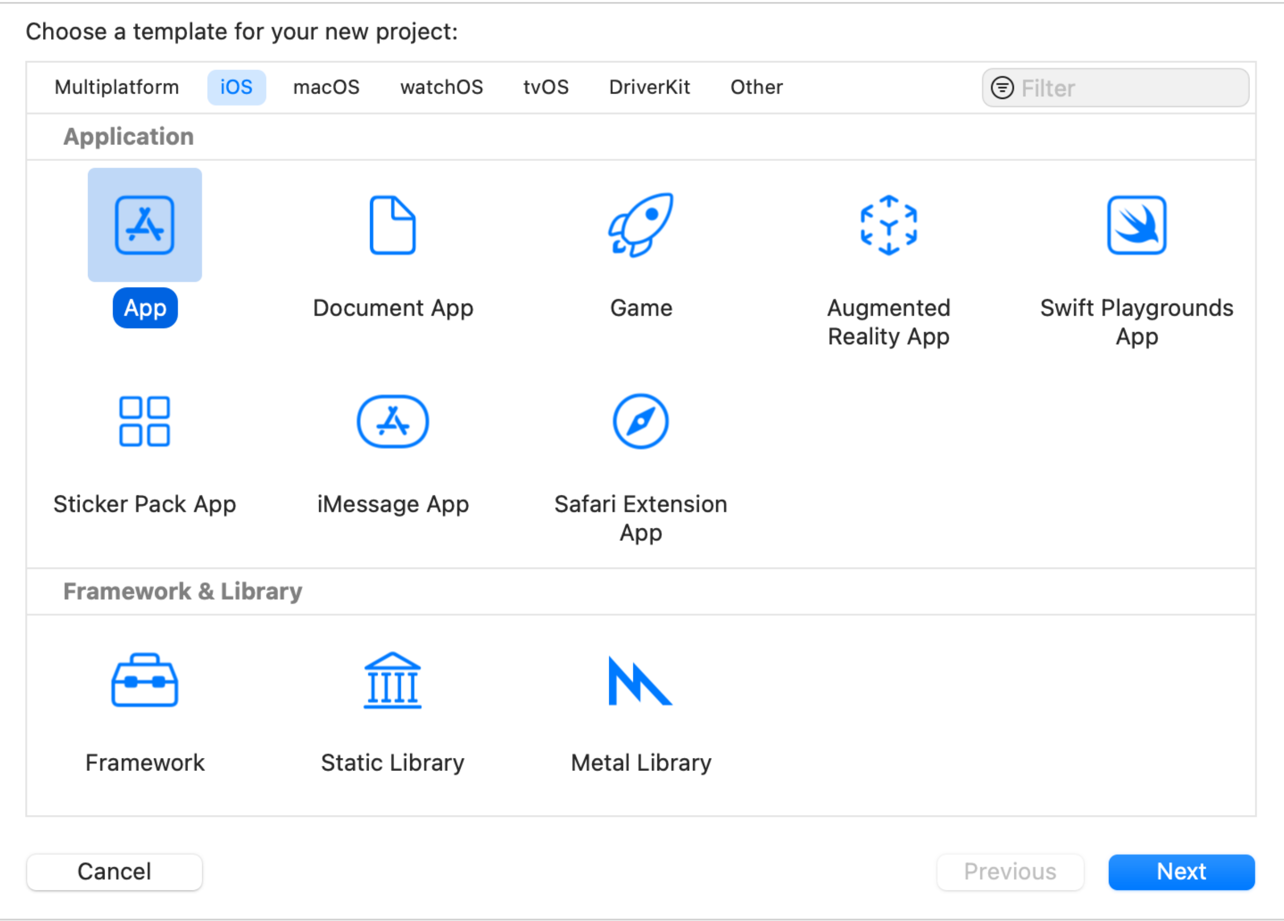
Once you've made your Android application cross-platform, you can create an iOS application and reuse the shared business logic in it.

1. Create an iOS project in Xcode.
2. Connect the framework to your iOS project.
3. Use the shared module from Swift.

Create an iOS project in Xcode

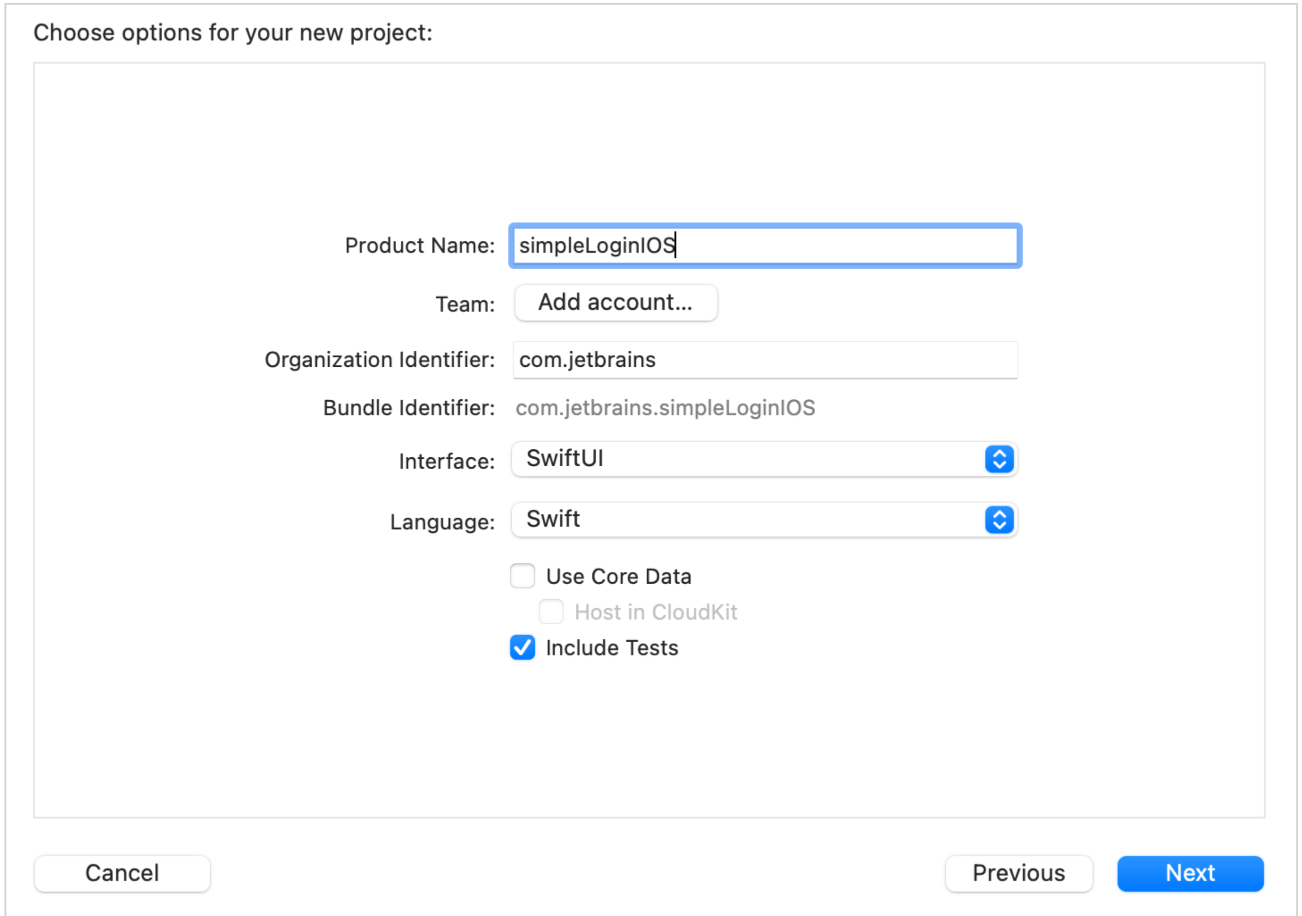
1. In Xcode, click File | New | Project.

2. Select a template for an iOS app and click Next.



iOS project template

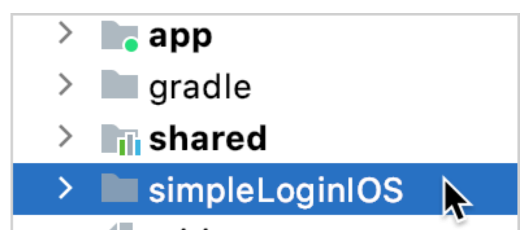
3. As the product name, specify simpleLoginIOS and click Next.



iOS project settings

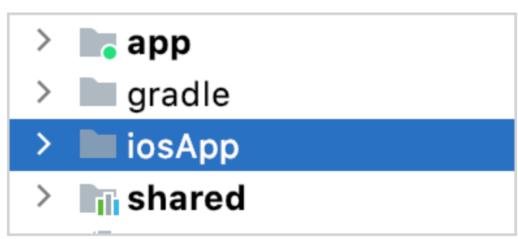
4. As the location for your project, select the directory that stores your cross-platform application, for example, kmm-integration-sample.

In Android Studio, you'll get the following structure:



iOS project in Android Studio

You can rename the simpleLoginIOS directory to iosApp for consistency with other top-level directories of your cross-platform project.



Renamed iOS project directory in Android Studio

Connect the framework to your iOS project

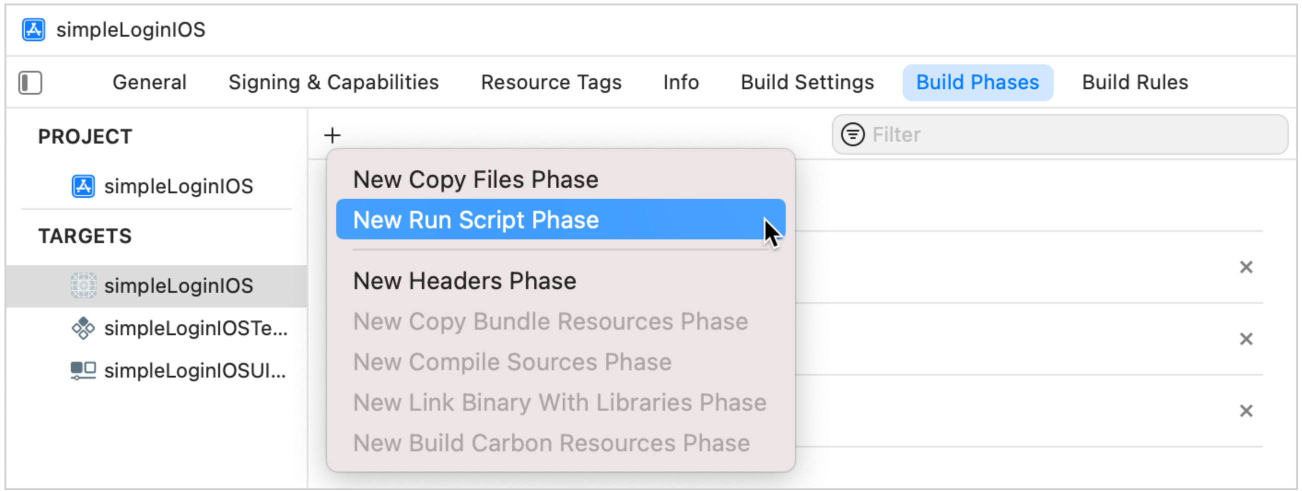
Once you have the framework, you can connect it to your iOS project manually.

> An alternative is to configure integration via CocoaPods, but that integration is beyond the scope of this tutorial.

Connect your framework to the iOS project manually:

1. In Xcode, open the iOS project settings by double-clicking the project name.

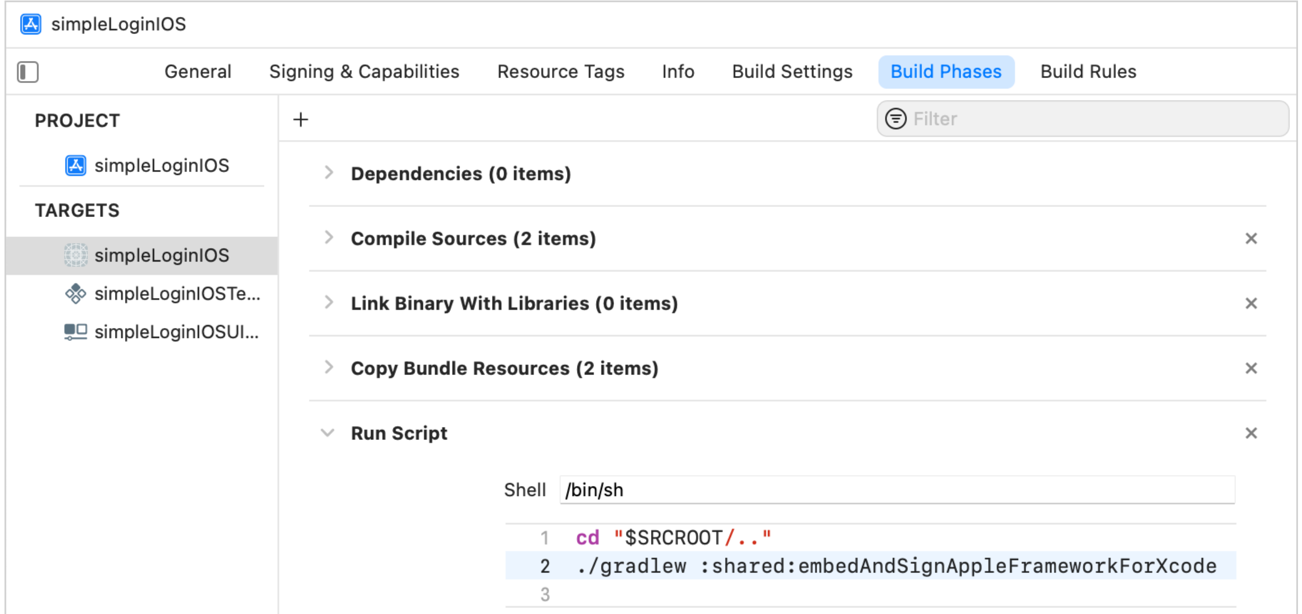
2. On the Build Phases tab of the project settings, click the + and add New Run Script Phase.



Add run script phase

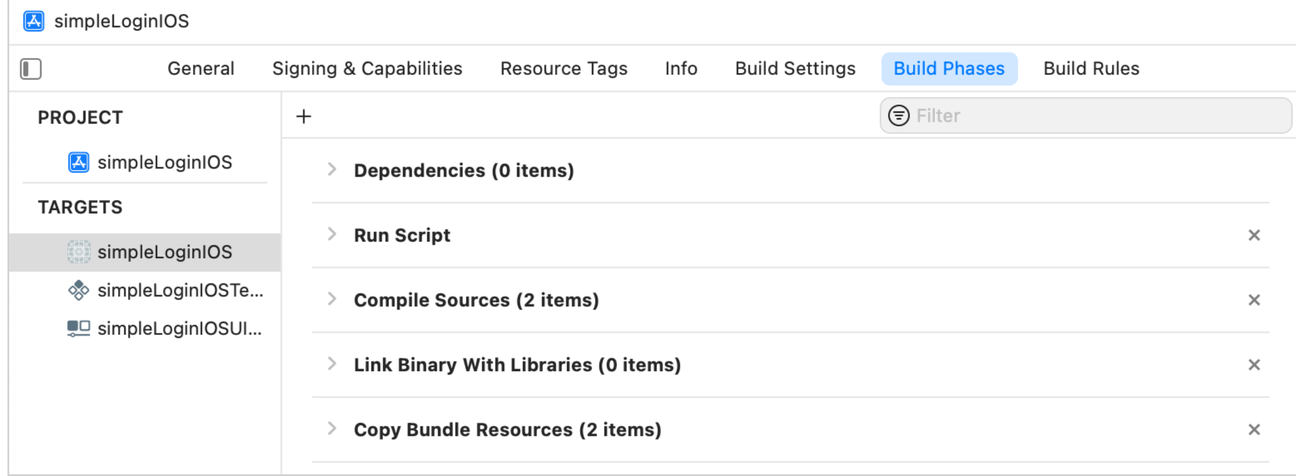
3. Add the following script:

```
cd "$SRCROOT/.."
./gradlew :shared:embedAndSignAppleFrameworkForXcode
```



Add the script

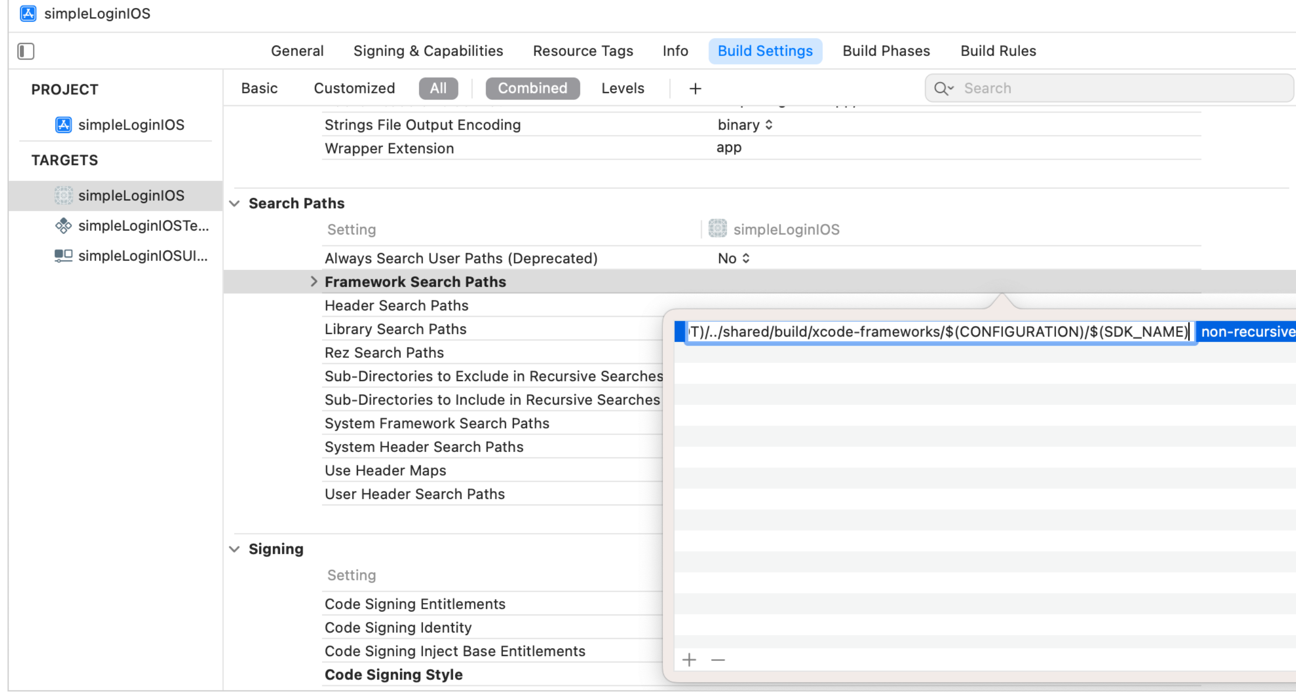
4. Move the Run Script phase before the Compile Sources phase.



Move the Run Script phase

5. On the Build Settings tab, switch to All build settings and specify the Framework Search Path under Search Paths:

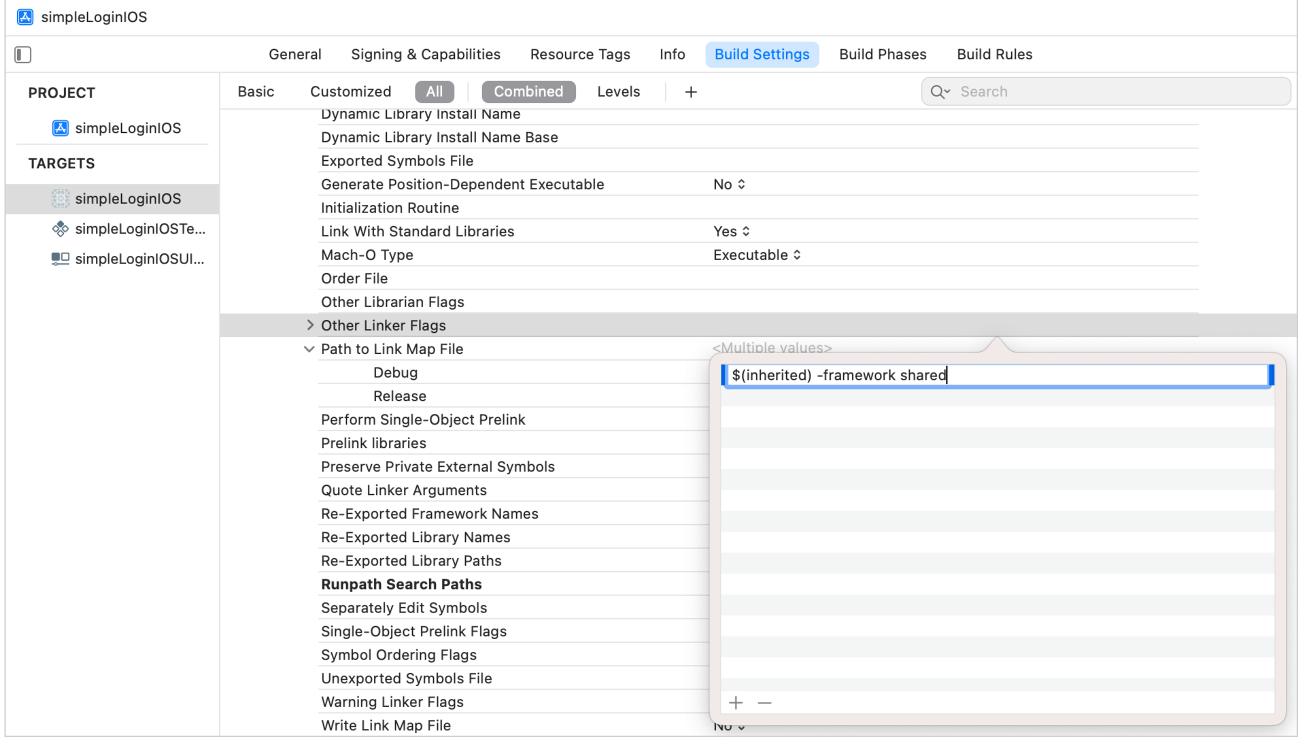
```
$(SRCROOT)/../shared/build/xcode-frameworks/$(CONFIGURATION)/$(SDK_NAME)
```



Framework search path

6. On the Build Settings tab, specify the Other Linker flags under Linking:

```
$(inherited) -framework shared
```



Linker flag

7. Build the project in Xcode. If everything is set up correctly, the project will successfully build.

> If you have a custom build configuration different from the default Debug or Release, on the Build Settings tab, add the KOTLIN_FRAMEWORK_BUILD_TYPE setting under User-Defined and set it to Debug or Release.

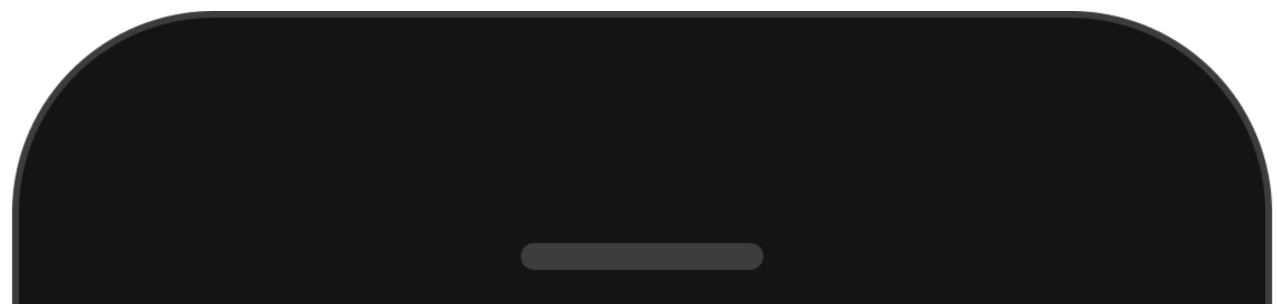
Use the shared module from Swift

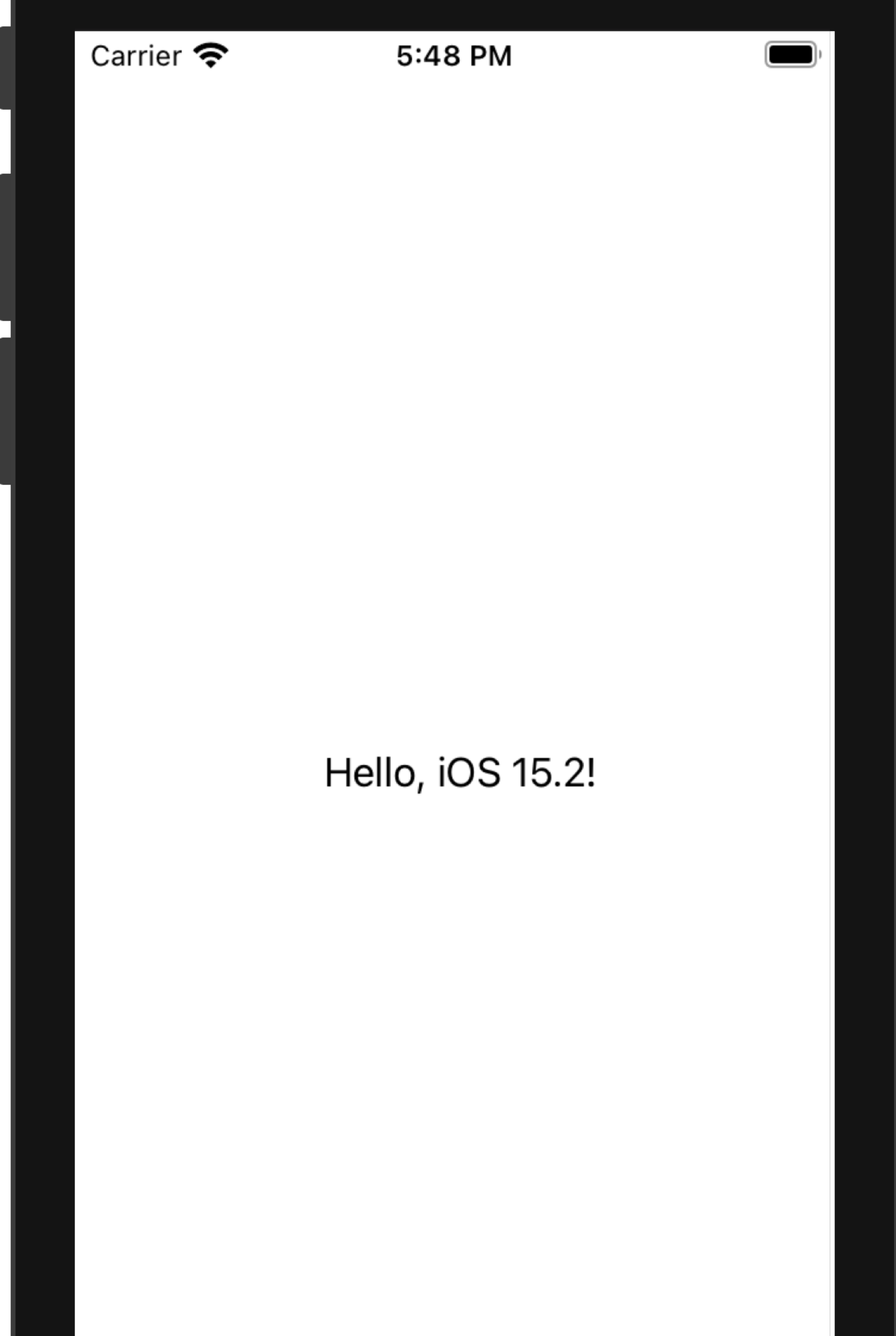
1. In Xcode, open the ContentView.swift file and import the shared module:

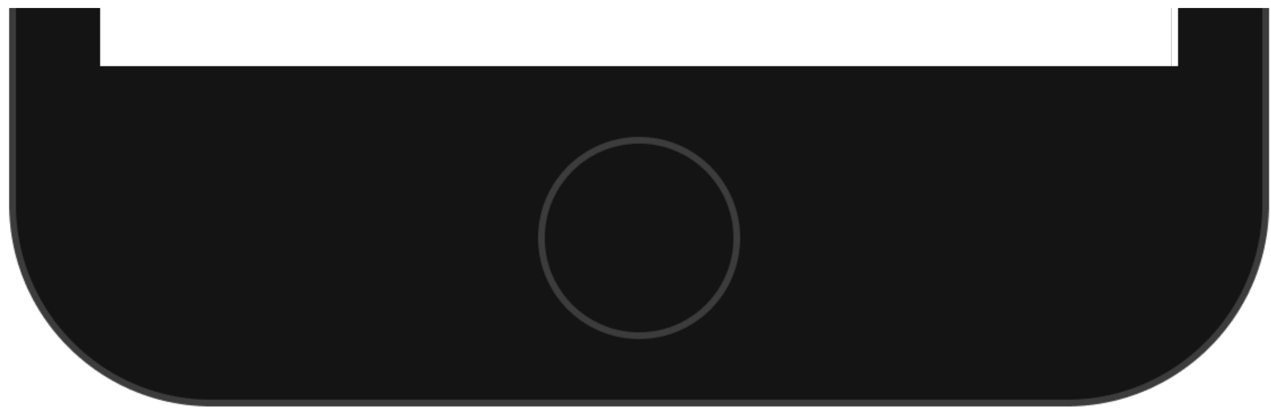
```
import shared
```

2. To check that it is properly connected, use the greet() function from the shared module of your cross-platform app:

```
import SwiftUI
import shared

struct ContentView: View {
    var body: some View {
        Text(Greeting().greet())
        .padding()
    }
}
```




Carrier
5:48 PM
Hello, iOS 15.2!



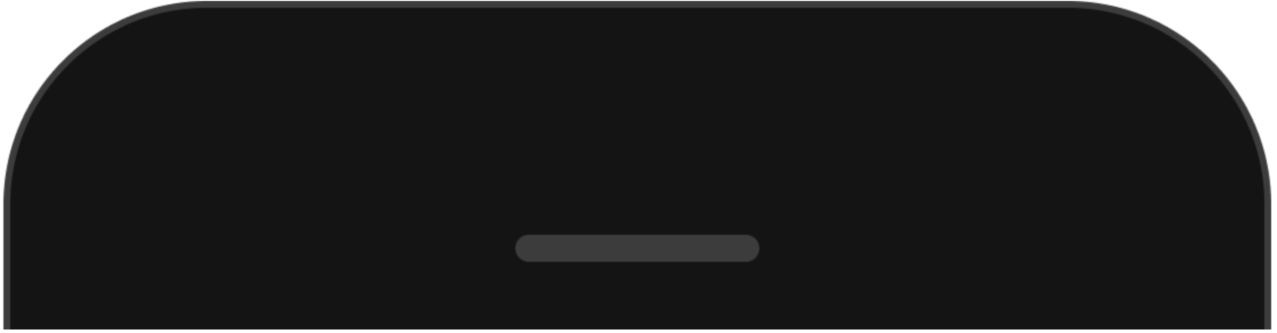
Greeting from the shared module

3. In ContentView.swift, write code for using data from the shared module and rendering the application UI:

   import SwiftUI import shared struct ContentView: View { @State private var username: String = "" @State private var password: String = "" @ObservedObject var viewModel: ContentView.ViewModel var body: some View { VStack(spacing: 15.0) { ValidatedTextField(titleKey: "Username", secured: false, text: $username, errorMessage: viewModel.formState.usernameError, onChange: { viewModel.loginDataChanged(username: username, password: password) }) ValidatedTextField(titleKey: "Password", secured: true, text: $password, errorMessage: viewModel.formState.passwordError, onChange: { viewModel.loginDataChanged(username: username, password: password) }) Button("Login") { viewModel.login(username: username, password: password) }.disabled(!viewModel.formState.isDataValid || (username.isEmpty && password.isEmpty)) } .padding(.all) } } struct ValidatedTextField: View { let titleKey: String let secured: Bool @Binding var text: String let errorMessage: String? let onChange: () -> () @ViewBuilder var textField: some View { if secured { SecureField(titleKey, text: $text) } else { TextField(titleKey, text: $text) } } var body: some View { ZStack { textField .textFieldStyle(RoundedBorderTextFieldStyle()) .autocapitalization(.none) .onChange(of: text) { _ in onChange() } if let errorMessage = errorMessage { HStack { Spacer() FieldTextErrorHint(error: errorMessage) }.padding(.horizontal, 5) } } } } struct FieldTextErrorHint: View { let error: String @State private var showingAlert = false var body: some View { Button(action: { self.showingAlert = true }) { Image(systemName: "exclamationmark.triangle.fill") .foregroundColor(.red) } .alert(isPresented: $showingAlert) { Alert(title: Text("Error"), message: Text(error), dismissButton: .default(Text("Got it!"))) } } } extension ContentView { struct LoginFormState { let usernameError: String? let passwordError: String? var isDataValid: Bool { get { return usernameError == nil && passwordError == nil } } } class ViewModel: ObservableObject { @Published var formState = LoginFormState(usernameError: nil, passwordError: nil) let loginValidator: LoginDataValidator let loginRepository: LoginRepository init(loginRepository: LoginRepository, loginValidator: LoginDataValidator) { self.loginRepository = loginRepository self.loginValidator = loginValidator } func login(username: String, password: String) { if let result = loginRepository.login(username: username, password: password) as? ResultSuccess { print("Successful login. Welcome, \(result.data.displayName)") } else { print("Error while logging in") } } func loginDataChanged(username: String, password: String) { formState = LoginFormState(usernameError: (loginValidator.checkUsername(username: username) as? LoginDataValidator.ResultError)?.message, passwordError: (loginValidator.checkPassword(password: password) as? LoginDataValidator.ResultError)?.message) } } }

4. In simpleLoginIOSApp.swift, import the shared module and specify the arguments for the ContentView() function:

```
import SwiftUI
import shared

@main
struct SimpleLoginIOSApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView(viewModel: .init(loginRepository: LoginRepository(dataSource: LoginDataSource()), loginValidator:
LoginDataValidator()))
        }
    }
}
```

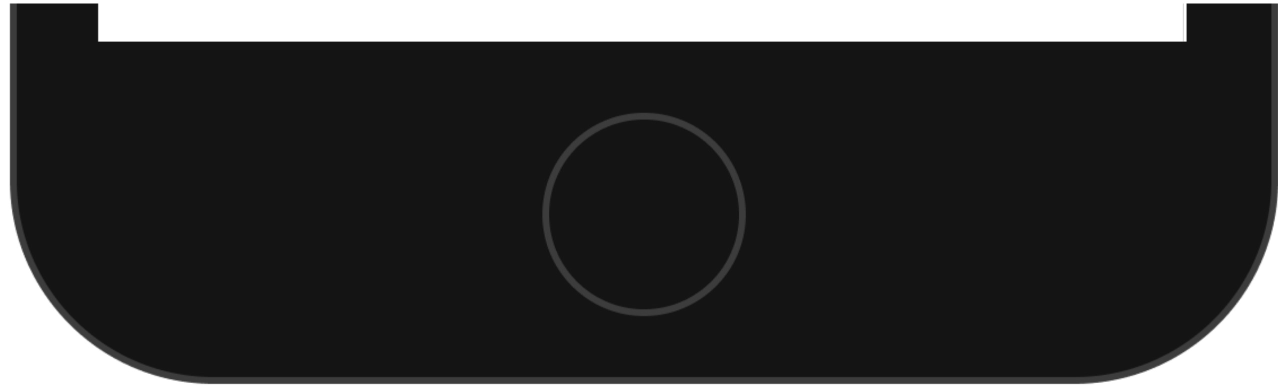


Carrier 6:29 PM

Jane

password

Login



Simple login application

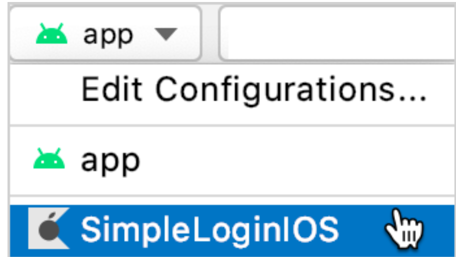
Enjoy the results – update the logic only once

Now your application is cross-platform. You can update the business logic in one place and see results on both Android and iOS.

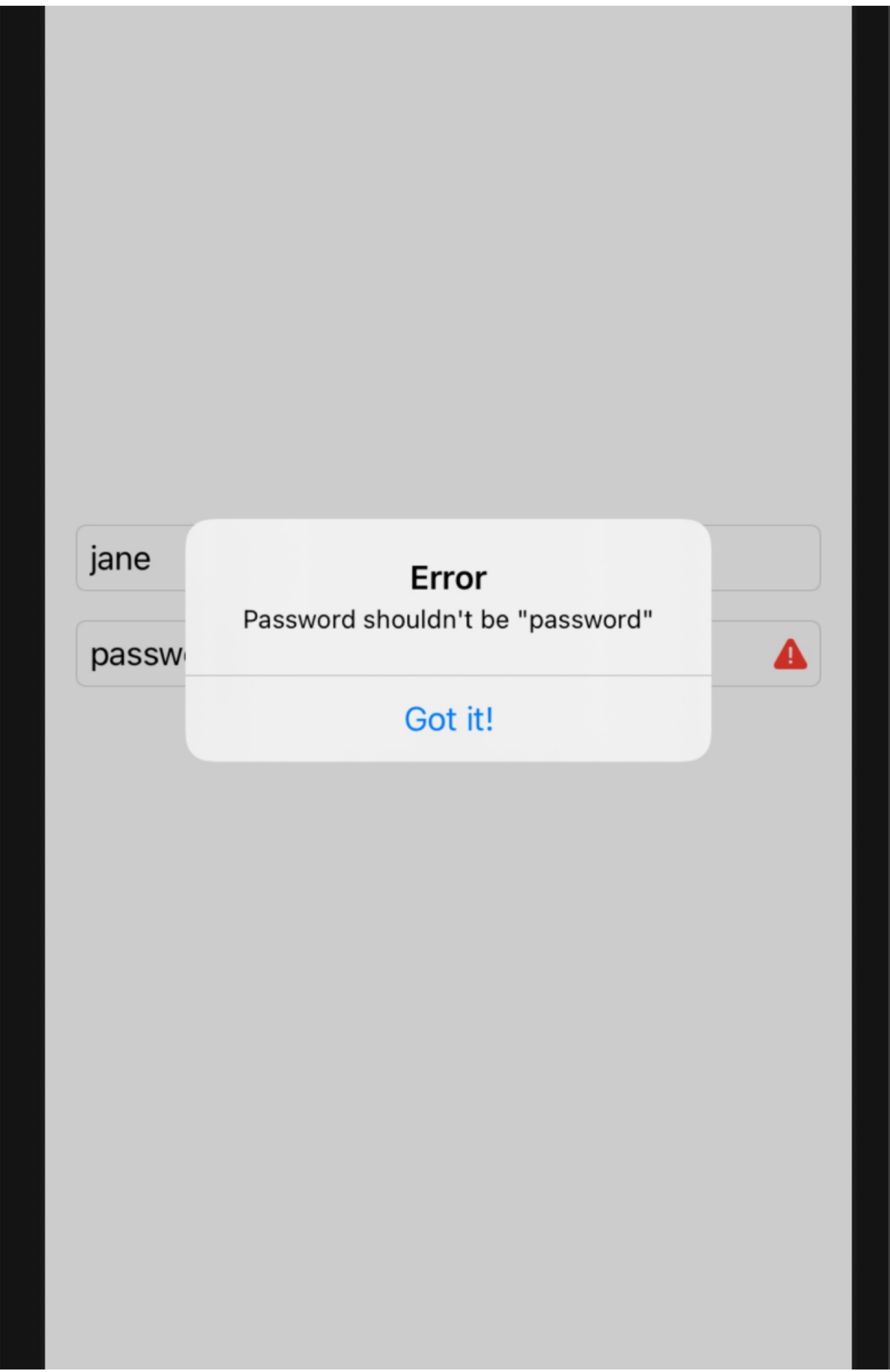
1. In Android Studio, change the validation logic for a user's password in the checkPassword() function of the LoginDataValidator class:

```
package com.jetbrains.simplelogin.shared.data

class LoginDataValidator {
//...
    fun checkPassword(password: String): Result {
        return when {
            password.length < 5 -> Result.Error("Password must be >5 characters")
            password.lowercase() == "password" -> Result.Error("Password shouldn't be \"password\"")
            else -> Result.Success
        }
    }
//...
}
```

2. Run both the iOS and Android applications from Android Studio to see the changes:



iOS run configuration


jane
passw
Error
Password shouldn't be "password"
Got it!

iOS application password error

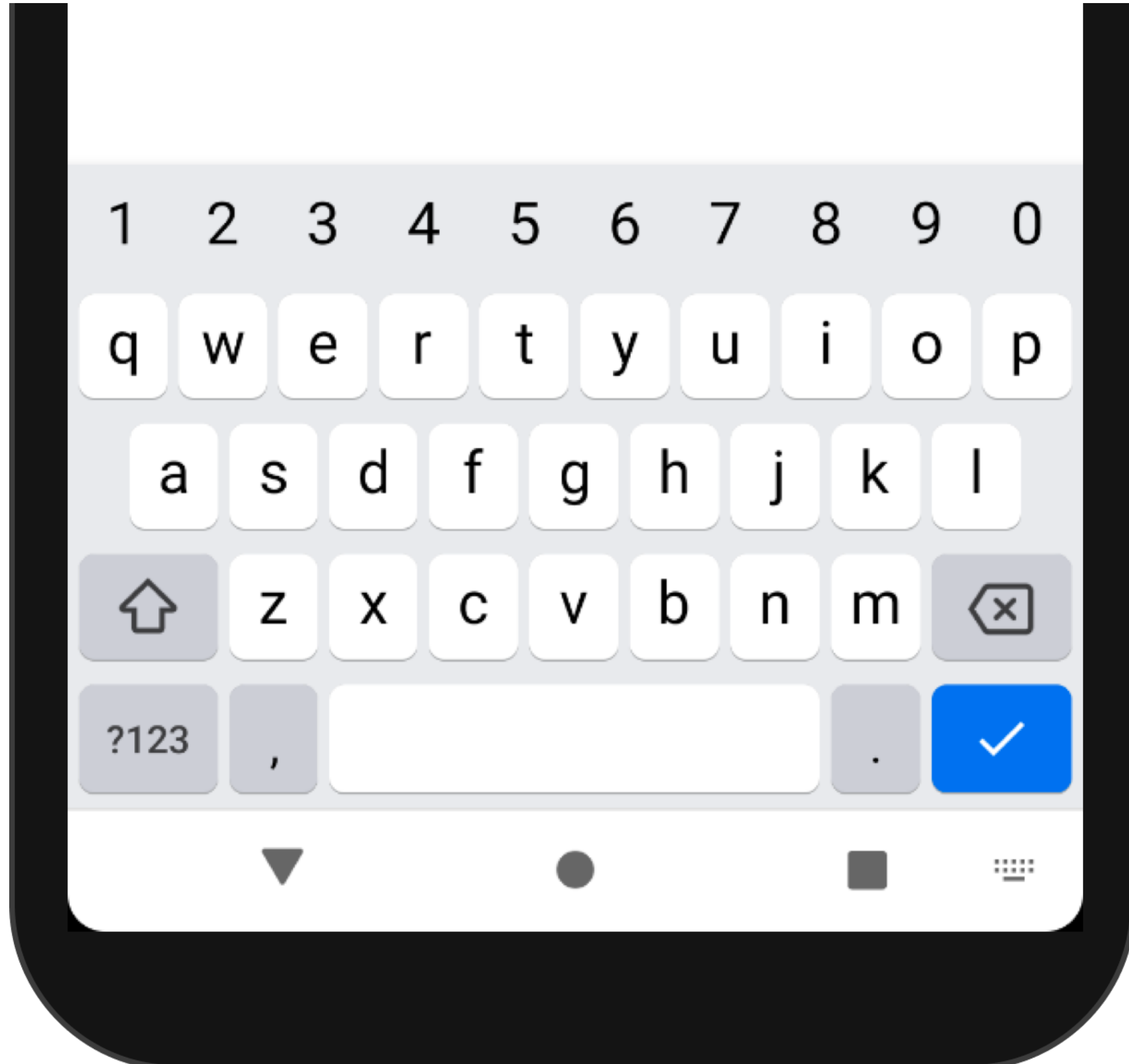
11:52

Simple Login

jane

Password shouldn't be "password"

SIGN IN OR REGISTER



Android application password error

You can review the final code for this tutorial.

What else to share?

You've shared the business logic of your application, but you can also decide to share other layers of your application. For example, the ViewModel class code is almost the same for Android and iOS applications, and you can share it if your mobile applications should have the same presentation layer.

What's next?

Once you've made your Android application cross-platform, you can move on and:

- Add dependencies on multiplatform libraries
- Add Android dependencies
- Add iOS dependencies

You can also check out community resources:

* Video: 3 ways to get your Kotlin JVM code ready for Kotlin Multiplatform Mobile

Publish your application

Once your mobile apps are ready for release, it's time to deliver them to the users by publishing them in app stores. Multiple stores are available for each platform. However, in this article we'll focus on the official ones: Google Play Store and Apple App Store. You'll learn how to prepare Kotlin Multiplatform applications for publishing, and we'll highlight the parts of this process that deserve special attention.

Android app

Since Kotlin is the main language for Android development, Kotlin Multiplatform has no obvious effect on compiling the project and building the Android app. Both the Android library produced from the shared module and the Android app itself are typical Android Gradle modules; they are no different from other Android libraries and apps. Thus, publishing the Android app from a Kotlin Multiplatform project is no different from the usual process described in the Android developer documentation.

iOS app

The iOS app from a Kotlin Multiplatform project is built from a typical Xcode project, so the main stages involved in publishing it are the same as described in the iOS developer documentation.

What is specific to Kotlin Multiplatform projects is compiling the shared Kotlin module into a framework and linking it to the Xcode project. Generally, all integration between the shared module and the Xcode project is done automatically by the Kotlin Multiplatform Mobile plugin for Android Studio. However, if you don't use the plugin, bear in mind the following when building and bundling the iOS project in Xcode:

* The shared Kotlin library compiles down to the native framework.
* You need to connect the framework compiled for the specific platform to the iOS app project.
* In the Xcode project settings, specify the path to the framework to search for the build system.
* After building the project, you should launch and test the app to make sure that there are no issues when working with the framework in runtime.

There are two ways you can connect the shared Kotlin module to the iOS project:

* Use the Kotlin/Native CocoaPods plugin, which allows you to use a multiplatform project with native targets as a CocoaPods dependency in your iOS project.
* Manually configure your Multiplatform project to create an iOS framework and the Xcode project to obtain its latest version. The Kotlin Multiplatform Mobile plugin for Android Studio usually does this configuration. Understand the project structure to implement it yourself.

Symbolicating crash reports

To help developers make their apps better, iOS provides a means for analyzing app crashes. For detailed crash analysis, it uses special debug symbol (.dSYM) files that match memory addresses in crash reports with locations in the source code, such as functions or line numbers.

By default, the release versions of iOS frameworks produced from the shared Kotlin module have an accompanying .dSYM file. This helps you analyze crashes that happen in the shared module's code.

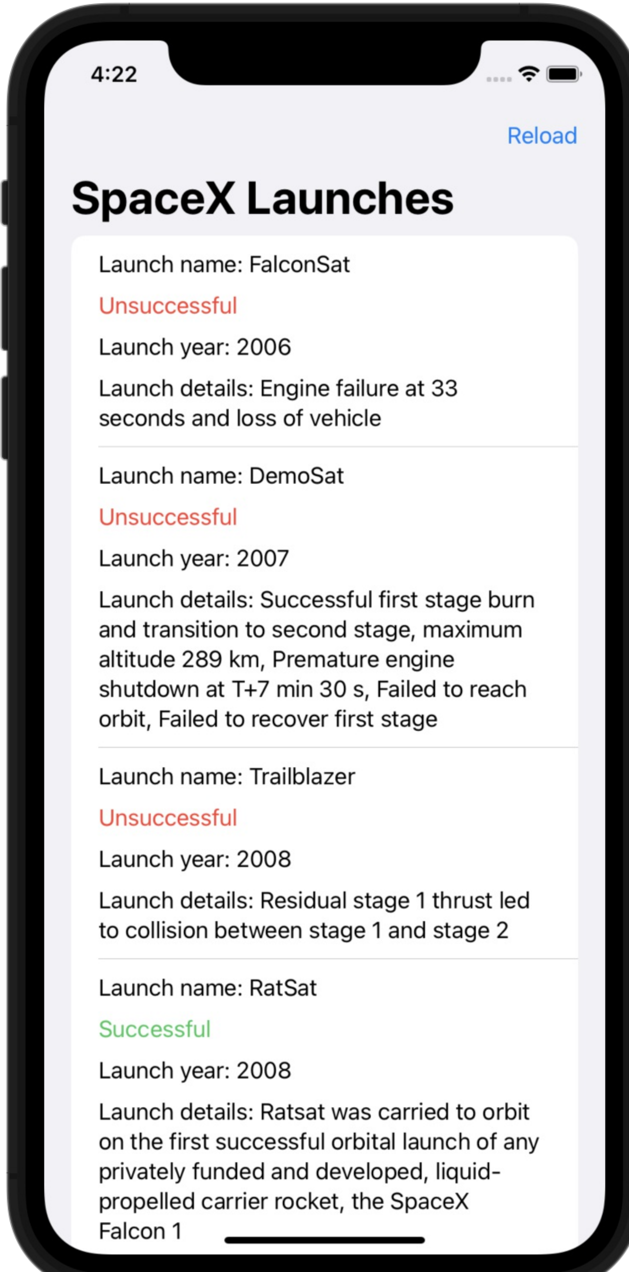
When an iOS app is rebuilt from bitcode, its dSYM file becomes invalid. For such cases, you can compile the shared module to a static framework that stores the debug information inside itself. For instructions on setting up crash report symbolication in binaries produced from Kotlin modules, see the Kotlin/Native documentation.

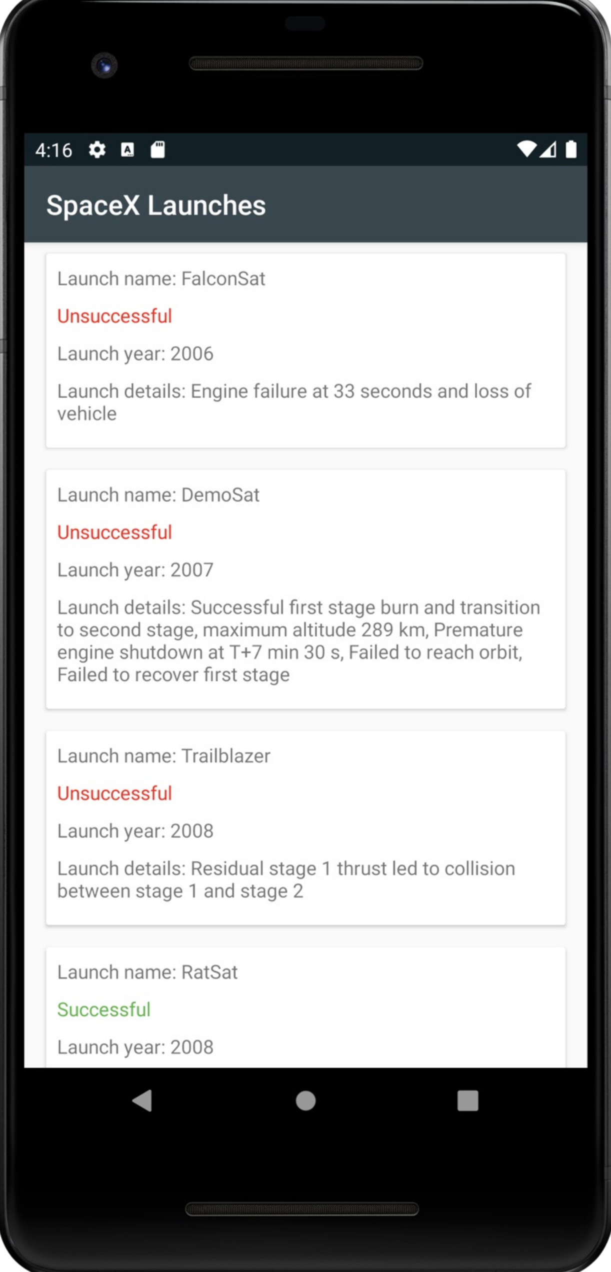
Create a multiplatform app using Ktor and SQLDelight – tutorial

This tutorial demonstrates how to use Android Studio to create a mobile application for iOS and Android using Kotlin Multiplatform Mobile with Ktor and SQLDelight.

The application will include a module with shared code for both the iOS and Android platforms. The business logic and data access layers will be implemented only once in the shared module, while the UI of both applications will be native.

The output will be an app that retrieves data over the internet from the public SpaceX API, saves it in a local database, and displays a list of SpaceX rocket launches together with the launch date, results, and a detailed description of the launch:





Emulator and Simulator

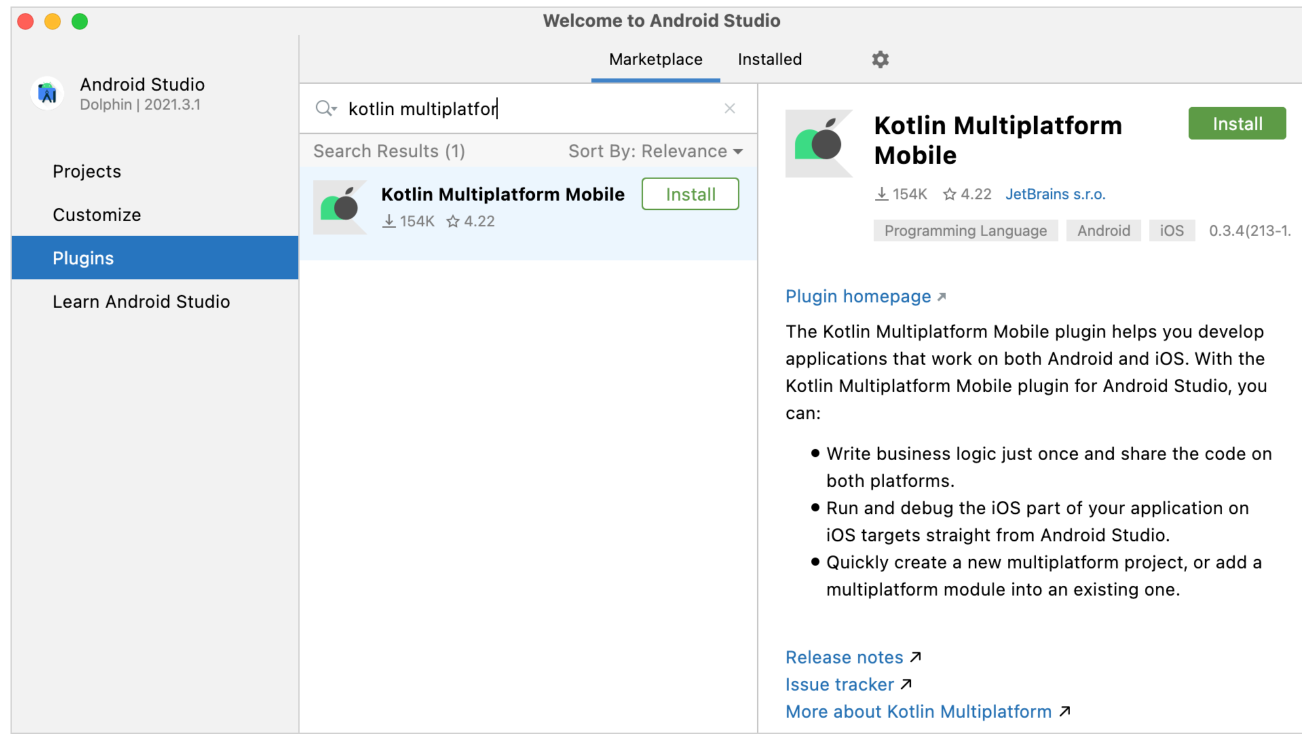
You will use the following multiplatform libraries in the project:

- Ktor as an HTTP client for retrieving data over the internet.
- kotlinx.serialization to deserialize JSON responses into objects of entity classes.
- kotlinx.coroutines to write asynchronous code.
- SQLDelight to generate Kotlin code from SQL queries and create a type-safe database API.

> You can find the template project as well as the source code of the final application on the corresponding GitHub repository.

Before you start

1. Download and install Android Studio.
2. Search for the Kotlin Multiplatform Mobile plugin in the Android Studio Marketplace and install it.



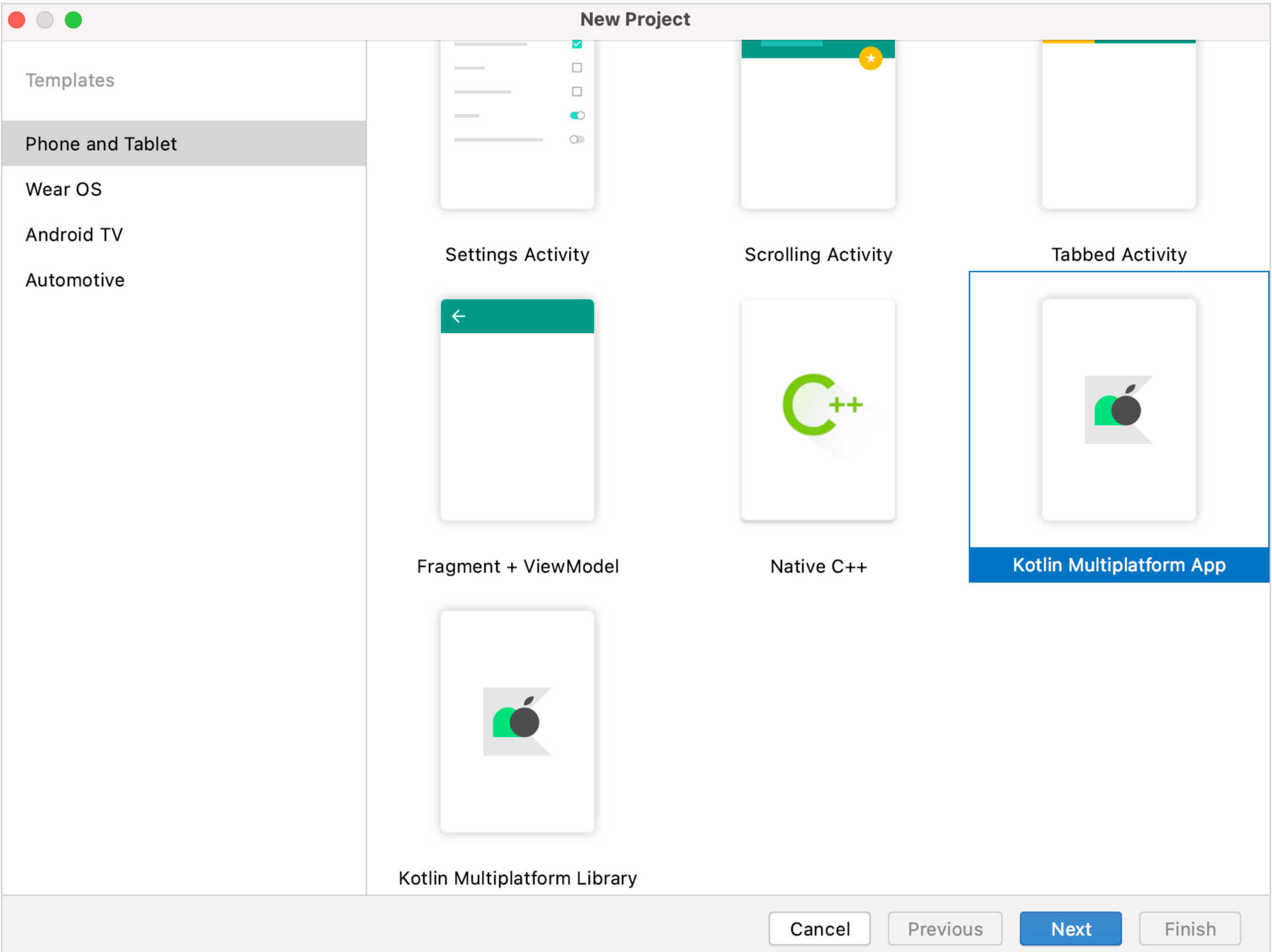
Kotlin Multiplatform Mobile plugin

3. Download and install Xcode.

For more details, see the Set up the environment section.

Create a Multiplatform project

1. In Android Studio, select File | New | New Project. In the list of project templates, select Kotlin Multiplatform App and then click Next.



Kotlin Multiplatform Mobile plugin wizard

2. Name your application and click Next.

3. Select Regular framework in the list of iOS framework distribution options.

New Project

Kotlin Multiplatform App

Creates a new Kotlin Multiplatform Mobile project that includes iOS and Android applications and a module with code shared on iOS and Android.

Android Application Name

androidApp

iOS Application Name

iosApp

Shared Module Name

shared

Add sample tests for Shared Module

iOS framework distribution

Regular framework

Select the most suitable type of Kotlin framework distribution to Xcode:

- **CocoaPods Dependency Manager:** Kotlin Multiplatform module is used as a CocoaPods dependency. Requires Ruby gem installation.
- **Regular framework:** Kotlin Multiplatform module is integrated via internal Gradle task and Xcode Build Steps configuration. Doesn't require 3rd party tools.

Learn more about CocoaPods integration.

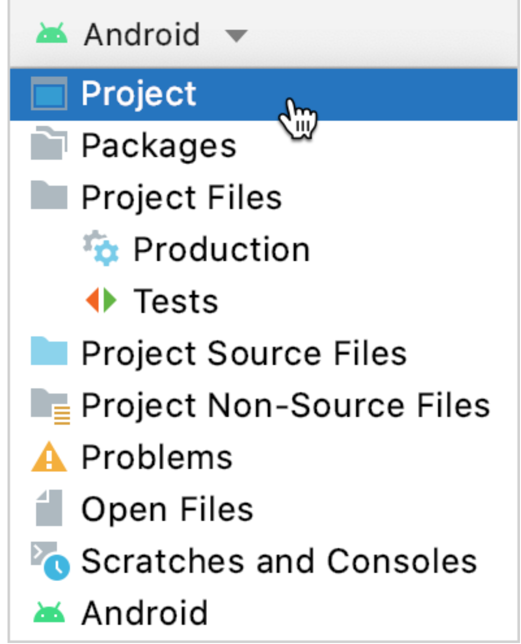
Learn more about Gradle packing task for iOS.

Cancel Previous Next Finish

Kotlin Multiplatform Mobile plugin wizard. Final step

4. Keep all other options default. Click Finish.

5. To view the complete structure of the multiplatform mobile project, switch the view from Android to Project.



Project view

For more on project features and how to use them, see Understand the project structure.

You can find the configured project on the master branch.

Add dependencies to the multiplatform library

To add a multiplatform library to the shared module, you need to add dependency instructions (implementation) for all libraries to the dependencies block of the relevant source sets in the build.gradle.kts file.

Both the kotlinx.serialization and SQLDelight libraries also require additional configurations.

1. In the shared directory, specify the dependencies on all the required libraries in the build.gradle.kts file:

```
val coroutinesVersion = "1.7.1"
val ktorVersion = "2.3.2"
val sqlDelightVersion = "1.5.5"
val dateTimeVersion = "0.4.0"

sourceSets {
    val commonMain by getting {
        dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutinesVersion")
            implementation("io.ktor:ktor-client-core:$ktorVersion")
            implementation("io.ktor:ktor-client-content-negotiation:$ktorVersion")
            implementation("io.ktor:ktor-serialization-kotlinx-json:$ktorVersion")
            implementation("com.squareup.sqldelight:runtime:$sqlDelightVersion")
            implementation("org.jetbrains.kotlinx:kotlinx-datetime:$dateTimeVersion")
            }
        }
    val androidMain by getting {
        dependencies {
            implementation("io.ktor:ktor-client-android:$ktorVersion")
            implementation("com.squareup.sqldelight:android-driver:$sqlDelightVersion")
        }
    }
    val iosMain by getting {
        // ...
        dependencies {
            implementation("io.ktor:ktor-client-darwin:$ktorVersion")
            implementation("com.squareup.sqldelight:native-driver:$sqlDelightVersion")
        }
    }
}
```

- Each library requires a core artifact in the common source set.
- Both the SQLDelight and Ktor libraries need platform drivers in the iOS and Android source sets, as well.
- In addition, Ktor needs the serialization feature to use kotlinx.serialization for processing network requests and responses.

2. At the very beginning of the build.gradle.kts file in the same shared directory, add the following lines to the plugins block:

```
    plugins {
    // ...
    kotlin("plugin.serialization") version "1.9.0"
    id("com.squareup.sqldelight")
}
```

3. Now go to the build.gradle.kts file in the project root directory and specify the classpath for the plugin in the build system dependencies:

```
buildscript {

    dependencies {
        // ...
        classpath("com.squareup.sqldelight:gradle-plugin:1.5.5")
    }
}
```

4. Finally, define the SQLDelight version in the gradle.properties file in the project root directory to ensure that the SQLDelight versions of the plugin and the libraries are the same:

```
sqlDelightVersion=1.5.5
```

5. Sync the Gradle project.

Learn more about adding dependencies on multiplatform libraries.

> You can find this state of the project on the final branch.

Create an application data model

The Kotlin Multiplatform app will contain the public SpaceXSDK class, the facade over networking and cache services. The application data model will have three entity classes with:

- General information about the launch
- A URL to external information
- Information about the rocket

1. In shared/src/commonMain/kotlin, add the com.jetbrains.handson.kmm.shared.entity package.
2. Create the Entity.kt file inside the package.
3. Declare all the data classes for basic entities:

   import kotlinx.datetime.TimeZone import kotlinx.datetime.toInstant import kotlinx.datetime.toLocalDateTime import kotlinx.serialization.SerialName import kotlinx.serialization.Serializable @Serializable data class RocketLaunch(@SerialName("flight_number") val flightNumber: Int, @SerialName("name") val missionName: String, @SerialName("date_utc") val launchDateUTC: String, @SerialName("details") val details: String?, @SerialName("success") val launchSuccess: Boolean?, @SerialName("links") val links: Links) { var launchYear = launchDateUTC.toInstant().toLocalDateTime(TimeZone.UTC).year } @Serializable data class Links(@SerialName("patch") val patch: Patch?, @SerialName("article") val article: String?) @Serializable data class Patch(@SerialName("small") val small: String?, @SerialName("large") val large: String?)

Each serializable class must be marked with the @Serializable annotation. The kotlinx.serialization plugin automatically generates a default serializer for @Serializable classes unless you explicitly pass a link to a serializer through the annotation argument.

However, you don't need to do that in this case. The @SerialName annotation allows you to redefine field names, which helps to declare properties in data classes with more easily readable names.

> You can find the state of the project after this section on the final branch.

Configure SQLDelight and implement cache logic

Configure SQLDelight

The SQLDelight library allows you to generate a type-safe Kotlin database API from SQL queries. During compilation, the generator validates the SQL queries and turns them into Kotlin code that can be used in the shared module.

The library is already in the project. To configure it, go to the shared directory and add the sqldelight block to the end of the build.gradle.kts file. The block will contain a list of databases and their parameters:

```
sqldelight {
    database("AppDatabase") {
        packageName = "com.jetbrains.handson.kmm.shared.cache"
    }
}
```

The packageName parameter specifies the package name for the generated Kotlin sources.

Consider installing the official SQLite plugin for working with .sq files.

Generate the database API

First, create the .sq file, which will contain all the needed SQL queries. By default, the SQLDelight plugin reads .sq from the sqldelight folder:

1. In shared/src/commonMain, create a new sqldelight directory and add the com.jetbrains.handson.kmm.shared.cache package.
2. Inside the package, create an .sq file with the name of the database, AppDatabase.sq. All the SQL queries for the application will be in this file.
3. The database will contain a table with data about launches. To create the table, add the following code to the AppDatabase.sq file:

```
CREATE TABLE Launch (
    flightNumber INTEGER NOT NULL,
    missionName TEXT NOT NULL,
    details TEXT,
    launchSuccess INTEGER AS Boolean DEFAULT NULL,
    launchDateUTC TEXT NOT NULL,
    patchUrlSmall TEXT,
    patchUrlLarge TEXT,
    articleUrl TEXT
);
```

4. To insert data into the tables, declare an SQL insert function:

```
insertLaunch:
INSERT INTO Launch(flightNumber, missionName, details, launchSuccess, launchDateUTC, patchUrlSmall, patchUrlLarge, articleUrl)
VALUES(?, ?, ?, ?, ?, ?, ?, ?);
```

5. To clear data in the tables, declare an SQL delete function:

```
removeAllLaunches:
DELETE FROM Launch;
```

6. In the same way, declare a function to retrieve data:

```
selectAllLaunchesInfo:
SELECT Launch.*
FROM Launch;
```

After the project is compiled, the generated Kotlin code will be stored in the shared/build/generated/sqldelight directory. The generator will create an interface named AppDatabase, as specified in build.gradle.kts.

Create platform database drivers

To initialize AppDatabase, pass an SqlDriver instance to it. SQLDelight provides multiple platform-specific implementations of the SQLite driver, so you need to create them for each platform separately. You can do this by using expected and actual declarations.

1. Create an abstract factory for database drivers. To do this, in shared/src/commonMain/kotlin, create the com.jetbrains.handson.kmm.shared.cache package and the DatabaseDriverFactory class inside it:

```kotlin
package com.jetbrains.handson.kmm.shared.cache

import com.squareup.sqldelight.db.SqlDriver

expect class DatabaseDriverFactory {
    fun createDriver(): SqlDriver
}
```

   Now provide actual implementations for this expected class.

2. On Android, the AndroidSqliteDriver class implements the SQLite driver. Pass the database information and the link to the context to the AndroidSqliteDriver class constructor.

   For this, in the shared/src/androidMain/kotlin directory, create the com.jetbrains.handson.kmm.shared.cache package and a DatabaseDriverFactory class inside it with the actual implementation:

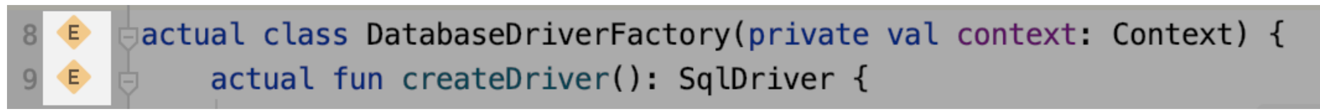
```
package com.jetbrains.handson.kmm.shared.cache

import android.content.Context
import com.squareup.sqldelight.android.AndroidSqliteDriver
import com.squareup.sqldelight.db.SqlDriver

actual class DatabaseDriverFactory(private val context: Context) {
    actual fun createDriver(): SqlDriver {
        return AndroidSqliteDriver(AppDatabase.Schema, context, "test.db")
    }
}
```

3. On iOS, the SQLite driver implementation is the NativeSqliteDriver class. In the shared/src/iosMain/kotlin directory, create a com.jetbrains.handson.kmm.shared.cache package and a DatabaseDriverFactory class inside it with the actual implementation:

```
package com.jetbrains.handson.kmm.shared.cache

import com.squareup.sqldelight.db.SqlDriver
import com.squareup.sqldelight.drivers.native.NativeSqliteDriver

actual class DatabaseDriverFactory {
    actual fun createDriver(): SqlDriver {
        return NativeSqliteDriver(AppDatabase.Schema, "test.db")
    }
}
```

Instances of these factories will be created later in the code of your Android and iOS projects.

You can navigate through the expect declarations and actual implementations by clicking the handy gutter icon:



Expect/Actual gutter

Implement cache

So far, you have added platform database drivers and an AppDatabase class to perform database operations. Now create a Database class, which will wrap the AppDatabase class and contain the caching logic.

1. In the common source set shared/src/commonMain/kotlin, create a new Database class in the com.jetbrains.handson.kmm.shared.cache package. It will be common to both platform logics.

2. To provide a driver for AppDatabase, pass an abstract DatabaseDriverFactory to the Database class constructor:

```
package com.jetbrains.handson.kmm.shared.cache

import com.jetbrains.handson.kmm.shared.entity.Links
import com.jetbrains.handson.kmm.shared.entity.Patch
import com.jetbrains.handson.kmm.shared.entity.RocketLaunch

internal class Database(databaseDriverFactory: DatabaseDriverFactory) {
    private val database = AppDatabase(databaseDriverFactory.createDriver())
    private val dbQuery = database.appDatabaseQueries
}
```

   This class's visibility is set to internal, which means it is only accessible from within the multiplatform module.

3. Inside the Database class, implement some data handling operations. Add a function to clear all the tables in the database in a single SQL transaction:

```
internal fun clearDatabase() {
    dbQuery.transaction {
        dbQuery.removeAllLaunches()
    }
 }
```

4. Create a function to get a list of all the rocket launches:

```kotlin
import com.jetbrains.handson.kmm.shared.entity.Links
import com.jetbrains.handson.kmm.shared.entity.Patch
import com.jetbrains.handson.kmm.shared.entity.RocketLaunch

internal fun getAllLaunches(): List<RocketLaunch> {
    return dbQuery.selectAllLaunchesInfo(::mapLaunchSelecting).executeAsList()
}

private fun mapLaunchSelecting(
    flightNumber: Long,
    missionName: String,
    details: String?,
    launchSuccess: Boolean?,
    launchDateUTC: String,
    patchUrlSmall: String?,
    patchUrlLarge: String?,
    articleUrl: String?
): RocketLaunch {
    return RocketLaunch(
        flightNumber = flightNumber.toInt(),
        missionName = missionName,
        details = details,
        launchDateUTC = launchDateUTC,
        launchSuccess = launchSuccess,
        links = Links(
            patch = Patch(
                small = patchUrlSmall,
                large = patchUrlLarge
            ),
            article = articleUrl
        )
    )
}
```

The argument passed to selectAllLaunchesInfo is a function that maps the database entity class to another type, which in this case is the RocketLaunch data model class.

5. Add a function to insert data into the database:

```kotlin
internal fun createLaunches(launches: List<RocketLaunch>) {
    dbQuery.transaction {
        launches.forEach { launch ->
            insertLaunch(launch)
        }
    }
}

private fun insertLaunch(launch: RocketLaunch) {
    dbQuery.insertLaunch(
        flightNumber = launch.flightNumber.toLong(),
        missionName = launch.missionName,
        details = launch.details,
        launchSuccess = launch.launchSuccess ?: false,
        launchDateUTC = launch.launchDateUTC,
        patchUrlSmall = launch.links.patch?.small,
        patchUrlLarge = launch.links.patch?.large,
        articleUrl = launch.links.article
    )
}
```

The Database class instance will be created later, along with the SDK facade class.

You can find the state of the project after this section on the final branch.

Implement an API service

To retrieve data over the internet, you'll need the SpaceX public API and a single method to retrieve the list of all launches from the v5/launches endpoint.

Create a class that will connect the application to the API:

1. In the common source set shared/src/commonMain/kotlin, create the com.jetbrains.handson.kmm.shared.network package and the SpaceXApi class inside it:

```kotlin
package com.jetbrains.handson.kmm.shared.network

import com.jetbrains.handson.kmm.shared.entity.RocketLaunch
import io.ktor.client.*
import io.ktor.client.call.*
import io.ktor.client.plugins.contentnegotiation.*
import io.ktor.client.request.*
import io.ktor.serialization.kotlinx.json.*
import kotlinx.serialization.json.Json

class SpaceXApi {
    private val httpClient = HttpClient {
        install(ContentNegotiation) {
            json(Json {
                ignoreUnknownKeys = true
                useAlternativeNames = false
            })
        }
    }
}
```

- This class executes network requests and deserializes JSON responses into entities from the entity package. The Ktor HttpClient instance initializes and stores the httpClient property.
- This code uses the Ktor ContentNegotiation plugin to deserialize the GET request result. The plugin processes the request and the response payload as JSON, serializing and deserializing them using a special serializer.

2. Declare the data retrieval function that will return the list of RocketLaunches:

```kotlin
suspend fun getAllLaunches(): List<RocketLaunch> {
    return httpClient.get("https://api.spacexdata.com/v5/launches").body()
}
```

- The getAllLaunches function has the suspend modifier because it contains a call of the suspend function get(), which includes an asynchronous operation to retrieve data over the internet and can only be called from a coroutine or another suspend function. The network request will be executed in the HTTP client's thread pool.
- The URL is defined inside the get() function to send requests.

Add internet access permission

To access the internet, the Android application needs the appropriate permission. Since all network requests are made from the shared module, adding the internet access permission to this module's manifest makes sense.

In the androidApp/src/main/AndroidManifest.xml file, add the following permission to the manifest:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.jetbrains.handson.androidApp">
    <uses-permission android:name="android.permission.INTERNET" />

</manifest>
```

You can find the state of the project after this section on the final branch.

Build an SDK

Your iOS and Android applications will communicate with the SpaceX API through the shared module, which will provide a public class.

1. In the com.jetbrains.handson.kmm.shared package of the common source set, create the SpaceXSDK class:

```kotlin
package com.jetbrains.handson.kmm.shared

import com.jetbrains.handson.kmm.shared.cache.Database
import com.jetbrains.handson.kmm.shared.cache.DatabaseDriverFactory
import com.jetbrains.handson.kmm.shared.network.SpaceXApi

class SpaceXSDK (databaseDriverFactory: DatabaseDriverFactory) {
    private val database = Database(databaseDriverFactory)
```

```
    private val api = SpaceXApi()
}
```

This class will be the facade over the Database and SpaceXApi classes.

2. To create a Database class instance, you'll need to provide the DatabaseDriverFactory platform instance to it, so you'll inject it from the platform code through the SpaceXSDK class constructor.

```
import com.jetbrains.handson.kmm.shared.entity.RocketLaunch

@Throws(Exception::class)
suspend fun getLaunches(forceReload: Boolean): List<RocketLaunch> {
    val cachedLaunches = database.getAllLaunches()
    return if (cachedLaunches.isNotEmpty() && !forceReload) {
        cachedLaunches
    } else {
        api.getAllLaunches().also {
            database.clearDatabase()
            database.createLaunches(it)
        }
    }
}
```

- The class contains one function for getting all launch information. Depending on the value of forceReload, it returns cached values or loads the data from the internet and then updates the cache with the results. If there is no cached data, it loads the data from the internet independently of the forceReload flag's value.
- Clients of your SDK could use a forceReload flag to load the latest information about the launches, which would allow the user to use the pull-to-refresh gesture.
- To handle exceptions produced by the Ktor client in Swift, the function is marked with the @Throws annotation.

All Kotlin exceptions are unchecked, while Swift has only checked errors. Thus, to make your Swift code aware of expected exceptions, Kotlin functions should be marked with the @Throws annotation specifying a list of potential exception classes.

> You can find the state of the project after this section on the final branch.

Create the Android application

The Kotlin Multiplatform Mobile plugin for Android Studio has already handled the configuration for you, so the Kotlin Multiplatform shared module is already connected to your Android application.

Before implementing the UI and the presentation logic, add all the required dependencies to the androidApp/build.gradle.kts:

```
// ...
dependencies {
    implementation(project(":shared"))
    implementation("com.google.android.material:material:1.9.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")
    implementation("androidx.swiperefreshlayout:swiperefreshlayout:1.1.0")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.1")
    implementation("androidx.core:core-ktx:1.10.1")
    implementation("androidx.recyclerview:recyclerview:1.3.0")
    implementation("androidx.cardview:cardview:1.0.0")
}
// ...
```

Implement the UI: display the list of rocket launches

1. To implement the UI, create the layout/activity_main.xml file in androidApp/src/main/res.

   The screen is based on the ConstraintLayout with the SwipeRefreshLayout inside it, which contains RecyclerView and FrameLayout with a background with a ProgressBar across its center:

   <?xml version="1.0" encoding="utf-8"?> <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android" xmlns:app="http://schemas.android.com/apk/res-auto" android:layout_width="match_parent" android:layout_height="match_parent">

<androidx.swiperefreshlayout.widget.SwipeRefreshLayout android:id="@+id/swipeContainer" android:layout_width="match_parent" android:layout_height="match_parent" app:layout_constraintBottom_toBottomOf="parent" app:layout_constraintEnd_toEndOf="parent" app:layout_constraintStart_toStartOf="parent" app:layout_constraintTop_toTopOf="parent"> <androidx.recyclerview.widget.RecyclerView android:id="@+id/launchesListRv" android:layout_width="match_parent" android:layout_height="match_parent" /> </androidx.swiperefreshlayout.widget.SwipeRefreshLayout> <FrameLayout android:id="@+id/progressBar" android:layout_width="0dp" android:layout_height="0dp" android:background="#fff" app:layout_constraintBottom_toBottomOf="parent" app:layout_constraintEnd_toEndOf="parent" app:layout_constraintStart_toStartOf="parent" app:layout_constraintTop_toTopOf="parent"> <ProgressBar android:layout_width="wrap_content" android:layout_height="wrap_content" android:layout_gravity="center" /> </FrameLayout> </androidx.constraintlayout.widget.ConstraintLayout>

2. In androidApp/src/main/java, replace the implementation of the MainActivity class, adding the properties for the UI elements:

```
class MainActivity : AppCompatActivity() {
    private lateinit var launchesRecyclerView: RecyclerView
    private lateinit var progressBarView: FrameLayout
    private lateinit var swipeRefreshLayout: SwipeRefreshLayout

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        title = "SpaceX Launches"
        setContentView(R.layout.activity_main)

        launchesRecyclerView = findViewById(R.id.launchesListRv)
        progressBarView = findViewById(R.id.progressBar)
        swipeRefreshLayout = findViewById(R.id.swipeContainer)
    }
}
```

3. For the RecyclerView element to work, you need to create an adapter (as a subclass of RecyclerView.Adapter) that will convert raw data into list item views. To do this, create a separate LaunchesRvAdapter class:

```
class LaunchesRvAdapter(var launches: List<RocketLaunch>) : RecyclerView.Adapter<LaunchesRvAdapter.LaunchViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): LaunchViewHolder {
        return LayoutInflater.from(parent.context)
            .inflate(R.layout.item_launch, parent, false)
            .run(::LaunchViewHolder)
    }

    override fun getItemCount(): Int = launches.count()

    override fun onBindViewHolder(holder: LaunchViewHolder, position: Int) {
        holder.bindData(launches[position])
    }

    inner class LaunchViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        // ...
        fun bindData(launch: RocketLaunch) {
            // ...
        }
    }
}
```

4. Create an item_launch.xml resource file in androidApp/src/main/res/layout/ with the items view layout:

<?xml version="1.0" encoding="utf-8"?> <androidx.cardview.widget.CardView xmlns:android="http://schemas.android.com/apk/res/android" xmlns:app="http://schemas.android.com/apk/res-auto" xmlns:card_view="http://schemas.android.com/tools" android:layout_width="match_parent" android:layout_height="wrap_content" android:layout_marginHorizontal="16dp" android:layout_marginVertical="8dp" card_view:cardCornerRadius="8dp"> <androidx.constraintlayout.widget.ConstraintLayout android:layout_width="match_parent" android:layout_height="wrap_content" android:paddingBottom="16dp"> <TextView android:id="@+id/missionName" android:layout_width="0dp" android:layout_height="wrap_content" android:layout_margin="8dp" app:layout_constraintEnd_toEndOf="parent" app:layout_constraintStart_toStartOf="parent" app:layout_constraintTop_toTopOf="parent" /> <TextView android:id="@+id/launchSuccess" android:layout_width="0dp" android:layout_height="wrap_content" android:layout_margin="8dp" app:layout_constraintEnd_toEndOf="parent" app:layout_constraintStart_toStartOf="parent" app:layout_constraintTop_toBottomOf="@+id/missionName" /> <TextView android:id="@+id/launchYear" android:layout_width="0dp" android:layout_height="wrap_content" android:layout_margin="8dp" app:layout_constraintEnd_toEndOf="parent" app:layout_constraintStart_toStartOf="parent" app:layout_constraintTop_toBottomOf="@+id/launchSuccess" /> <TextView android:id="@+id/details" android:layout_width="0dp" android:layout_height="wrap_content" android:layout_margin="8dp" app:layout_constraintEnd_toEndOf="parent" app:layout_constraintStart_toStartOf="parent" app:layout_constraintTop_toBottomOf="@+id/launchYear" /> </androidx.constraintlayout.widget.ConstraintLayout> </androidx.cardview.widget.CardView>

5. In androidApp/src/main/res/values/, either create your appearance of the app or copy the following styles:

colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#37474f</color>
    <color name="colorPrimaryDark">#102027</color>
    <color name="colorAccent">#62727b</color>

    <color name="colorSuccessful">#4BB543</color>
    <color name="colorUnsuccessful">#FC100D</color>
    <color name="colorNoData">#615F5F</color>
</resources>
```

strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">SpaceLaunches</string>

    <string name="successful">Successful</string>
    <string name="unsuccessful">Unsuccessful</string>
    <string name="no_data">No data</string>

    <string name="launch_year_field">Launch year: %s</string>
    <string name="mission_name_field">Launch name: %s</string>
    <string name="launch_success_field">Launch success: %s</string>
    <string name="details_field">Launch details: %s</string>
</resources>
```

styles.xml

```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

6. Complete the implementation of the RecyclerView.Adapter:

```
class LaunchesRvAdapter(var launches: List<RocketLaunch>) : RecyclerView.Adapter<LaunchesRvAdapter.LaunchViewHolder>() {
    // ...
    inner class LaunchViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        private val missionNameTextView = itemView.findViewById<TextView>(R.id.missionName)
        private val launchYearTextView = itemView.findViewById<TextView>(R.id.launchYear)
        private val launchSuccessTextView = itemView.findViewById<TextView>(R.id.launchSuccess)
        private val missionDetailsTextView = itemView.findViewById<TextView>(R.id.details)

        fun bindData(launch: RocketLaunch) {
            val ctx = itemView.context
            missionNameTextView.text = ctx.getString(R.string.mission_name_field, launch.missionName)
            launchYearTextView.text = ctx.getString(R.string.launch_year_field, launch.launchYear.toString())
            missionDetailsTextView.text = ctx.getString(R.string.details_field, launch.details ?: "")
            val launchSuccess = launch.launchSuccess
            if (launchSuccess != null ) {
                if (launchSuccess) {
                    launchSuccessTextView.text = ctx.getString(R.string.successful)
                    launchSuccessTextView.setTextColor((ContextCompat.getColor(itemView.context, R.color.colorSuccessful)))
                } else {
                    launchSuccessTextView.text = ctx.getString(R.string.unsuccessful)
                    launchSuccessTextView.setTextColor((ContextCompat.getColor(itemView.context, R.color.colorUnsuccessful)))
                }
            } else {
                launchSuccessTextView.text = ctx.getString(R.string.no_data)
                launchSuccessTextView.setTextColor((ContextCompat.getColor(itemView.context, R.color.colorNoData)))
            }
        }
    }
}
```

7. Update the MainActivity class as follows:

```
class MainActivity : AppCompatActivity() {
    // ...
    private val launchesRvAdapter = LaunchesRvAdapter(listOf())

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        launchesRecyclerView.adapter = launchesRvAdapter
        launchesRecyclerView.layoutManager = LinearLayoutManager(this)

        swipeRefreshLayout.setOnRefreshListener {
            swipeRefreshLayout.isRefreshing = false
            displayLaunches(true)
        }

        displayLaunches(false)
    }

    private fun displayLaunches(needReload: Boolean) {
        // TODO: Presentation logic
    }
}
```

Here you create an instance of LaunchesRvAdapter, configure the RecyclerView component, and implement all the LaunchesListView interface functions. To catch the screen refresh gesture, you add a listener to the SwipeRefreshLayout.

Implement the presentation logic

1. Create an instance of the SpaceXSDK class from the shared module and inject an instance of DatabaseDriverFactory in it:

```
class MainActivity : AppCompatActivity() {
    // ...
    private val sdk = SpaceXSDK(DatabaseDriverFactory(this))
}
```

2. Implement the private function displayLaunches(needReload: Boolean). It runs the getLaunches() function inside the coroutine launched in the main CoroutineScope, handles exceptions, and displays the error text in the toast message:

```
class MainActivity : AppCompatActivity() {
    private val mainScope = MainScope()
    // ...
    override fun onDestroy() {
        super.onDestroy()
        mainScope.cancel()
    }
    // ...
    private fun displayLaunches(needReload: Boolean) {
        progressBarView.isVisible = true
        mainScope.launch {
            kotlin.runCatching {
                sdk.getLaunches(needReload)
            }.onSuccess {
                launchesRvAdapter.launches = it
                launchesRvAdapter.notifyDataSetChanged()
            }.onFailure {
                Toast.makeText(this@MainActivity, it.localizedMessage, Toast.LENGTH_SHORT).show()
            }
            progressBarView.isVisible = false
        }
    }
}
```

3. Select androidApp from the run configurations menu, choose an emulator, and click the run button:

4:16

SpaceX Launches

Launch name: FalconSat

Unsuccessful

Launch year: 2006

Launch details: Engine failure at 33 seconds and loss of vehicle

Launch name: DemoSat

Unsuccessful

Launch year: 2007

Launch details: Successful first stage burn and transition to second stage, maximum altitude 289 km, Premature engine shutdown at T+7 min 30 s, Failed to reach orbit, Failed to recover first stage

Launch name: Trailblazer

Unsuccessful

Launch year: 2008

Launch details: Residual stage 1 thrust led to collision between stage 1 and stage 2

Launch name: RatSat

Android application

You've just created an Android application that has its business logic implemented in the Kotlin Multiplatform Mobile module.

> You can find the state of the project after this section on the final branch.

Create the iOS application

For the iOS part of the project, you'll make use of SwiftUI to build the user interface and the "Model View View-Model" pattern to connect the UI to the shared module, which contains all the business logic.

The shared module is already connected to the iOS project because the Android Studio plugin wizard has done all the configuration. You can import it the same way you would regular iOS dependencies: import shared.

Implement the UI

First, you'll create a RocketLaunchRow SwiftUI view for displaying an item from the list. It will be based on the HStack and VStack views. There will be extensions on the RocketLaunchRow structure with useful helpers for displaying the data.

1. Launch your Xcode app and select Open a project or file.
2. Navigate to your project and select the iosApp folder. Click Open.
3. In your Xcode project, create a new Swift file with the type SwiftUI View, name it RocketLaunchRow, and update it with the following code:

```
import SwiftUI
import shared

struct RocketLaunchRow: View {
    var rocketLaunch: RocketLaunch

    var body: some View {
        HStack() {
            VStack(alignment: .leading, spacing: 10.0) {
                Text("Launch name: \(rocketLaunch.missionName)")
                Text(launchText).foregroundColor(launchColor)
                Text("Launch year: \(String(rocketLaunch.launchYear))")
                Text("Launch details: \(rocketLaunch.details ?? "")")
            }
            Spacer()
        }
    }
}
```

```
extension RocketLaunchRow {
    private var launchText: String {
        if let isSuccess = rocketLaunch.launchSuccess {
            return isSuccess.boolValue ? "Successful" : "Unsuccessful"
        } else {
            return "No data"
        }
    }

    private var launchColor: Color {
        if let isSuccess = rocketLaunch.launchSuccess {
            return isSuccess.boolValue ? Color.green : Color.red
        } else {
            return Color.gray
        }
    }
}
```

The list of launches will be displayed in the ContentView, which the project wizard has already created.

4. Create a ViewModel class for the ContentView, which will prepare and manage the data. Declare it as an extension to the ContentView, as they are closely connected, and then add the following code to ContentView.swift:

```
// ...
extension ContentView {
    enum LoadableLaunches {
        case loading
        case result([RocketLaunch])
        case error(String)
    }

   @MainActor
   class ViewModel: ObservableObject {
       @Published var launches = LoadableLaunches.loading
   }
}
```

- The Combine framework connects the view model (ContentView.ViewModel) with the view (ContentView).
- ContentView.ViewModel is declared as an ObservableObject and @Published wrapper is used for the launches property, so the view model will emit signals whenever this property changes.

5. Implement the body of the ContentView file and display the list of launches:

```
struct ContentView: View {
 @ObservedObject private(set) var viewModel: ViewModel

     var body: some View {
         NavigationView {
             listView()
             .navigationBarTitle("SpaceX Launches")
             .navigationBarItems(trailing:
                 Button("Reload") {
                     self.viewModel.loadLaunches(forceReload: true)
             })
         }
     }

     private func listView() -> AnyView {
         switch viewModel.launches {
         case .loading:
             return AnyView(Text("Loading...").multilineTextAlignment(.center))
         case .result(let launches):
             return AnyView(List(launches) { launch in
                 RocketLaunchRow(rocketLaunch: launch)
             })
         case .error(let description):
             return AnyView(Text(description).multilineTextAlignment(.center))
         }
     }
 }
```

The @ObservedObject property wrapper is used to subscribe to the view model.

6. To make it compile, the RocketLaunch class needs to confirm the Identifiable protocol, as it is used as a parameter for initializing the List Swift UIView. The

RocketLaunch class already has a property named id, so add the following to the bottom of ContentView.swift:

```
extension RocketLaunch: Identifiable { }
```

Load the data

To retrieve the data about the rocket launches in the view model, you'll need an instance of SpaceXSDK from the Multiplatform library.

1. In ContentView.swift, pass it in through the constructor:

```
extension ContentView {
    // ...
    @MainActor
    class ViewModel: ObservableObject {
        let sdk: SpaceXSDK
        @Published var launches = LoadableLaunches.loading

        init(sdk: SpaceXSDK) {
            self.sdk = sdk
            self.loadLaunches(forceReload: false)
        }

        func loadLaunches(forceReload: Bool) {
            // TODO: retrieve data
        }
    }
}
```

2. Call the getLaunches() function from the SpaceXSDK class and save the result in the launches property:

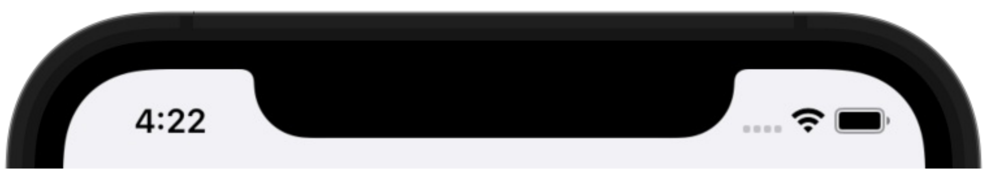
```
func loadLaunches(forceReload: Bool) {
    Task {
        do {
            self.launches = .loading
            let launches = try await sdk.getLaunches(forceReload: forceReload)
            self.launches = .result(launches)
        } catch {
            self.launches = .error(error.localizedDescription)
        }
    }
}
```

- When you compile a Kotlin module into an Apple framework, suspending functions are available in it as Swift's async/await mechanism.
- Since the getLaunches function is marked with the @Throws(Exception::class) annotation, any exceptions that are instances of the Exception class or its subclass will be propagated as NSError. Therefore, all such errors can be caught by the loadLaunches() function.

3. Go to the entry point of the app, iOSApp.swift, and initialize the SDK, view, and view model:

```
import SwiftUI
import shared

@main
struct iOSApp: App {
    let sdk = SpaceXSDK(databaseDriverFactory: DatabaseDriverFactory())
    var body: some Scene {
        WindowGroup {
            ContentView(viewModel: .init(sdk: sdk))
        }
    }
}
```

4. In Android Studio, switch to the iosApp configuration, choose an emulator, and run it to see the result:



Reload

SpaceX Launches

Launch name: FalconSat

Unsuccessful

Launch year: 2006

Launch details: Engine failure at 33 seconds and loss of vehicle

Launch name: DemoSat

Unsuccessful

Launch year: 2007

Launch details: Successful first stage burn and transition to second stage, maximum altitude 289 km, Premature engine shutdown at T+7 min 30 s, Failed to reach orbit, Failed to recover first stage

Launch name: Trailblazer

Unsuccessful

Launch year: 2008

Launch details: Residual stage 1 thrust led to collision between stage 1 and stage 2

Launch name: RatSat

Successful

Launch year: 2008

Launch details: Ratsat was carried to orbit on the first successful orbital launch of any privately funded and developed, liquid-propelled carrier rocket, the SpaceX Falcon 1

iOS Application

You can find the final version of the project on the final branch.

What's next?

This tutorial features some potentially resource-heavy operations, like parsing JSON and making requests to the database in the main thread. To learn about how to write concurrent code and optimize your app, see How to work with concurrency.

You can also check out these additional learning materials:

- Use the Ktor HTTP client in multiplatform projects
- Make your Android application work on iOS
- Introduce your team to Kotlin Multiplatform Mobile

Get started with Kotlin Multiplatform

Kotlin Multiplatform is in Beta. It is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make.

Support for multiplatform programming is one of Kotlin's key benefits. It reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming.

Learn more about Kotlin Multiplatform benefits.

Start from scratch

- Create and publish a multiplatform library. Complete a project for JVM, web, and native platforms, which can be used from any other common code (for example, shared with Android and iOS). Learn how to write tests that can be executed on all platforms and use an efficient implementation provided by a specific platform.
- Get started with Kotlin Multiplatform for mobile. Create your first cross-platform application that works on Android and iOS with the help of the Kotlin Multiplatform Mobile plugin for Android Studio. Learn how to create, run, and add dependencies to multiplatform mobile applications.
- Share UIs between iOS and Android. Create a Kotlin Multiplatform application that uses the Compose Multiplatform UI framework for sharing UIs between iOS and Android.

Dive deep into Kotlin Multiplatform

Once you have gained some experience with Kotlin Multiplatform and want to know how to solve particular cross-platform development tasks:

- Share code on platforms in your Kotlin Multiplatform project.
- Connect to platform-specific APIs using the Kotlin mechanism of expected and actual declarations.
- Set up targets manually for your Kotlin Multiplatform project.
- Add dependencies on the standard, test, or another kotlinx library.
- Configure compilations for production and test purposes in your project.
- Run tests for JVM, JavaScript, Android, Linux, Windows, macOS, iOS, watchOS, and tvOS simulators.
- Publish a multiplatform library to the Maven repository.
- Build native binaries as executables or shared libraries, like universal frameworks or XCFrameworks.

Get help

- Kotlin Slack: Get an invite and join the #multiplatform channel
- StackOverflow: Subscribe to the "kotlin-multiplatform" tag
- Kotlin issue tracker: Report a new issue

Understand Multiplatform project structure

Discover main parts of your multiplatform project:

- Multiplatform plugin
- Targets
- Source sets
- Compilations

Multiplatform plugin

When you create a multiplatform project, the Project Wizard automatically applies the kotlin-multiplatform Gradle plugin in the file build.gradle(.kts).

You can also apply it manually.

> The kotlin-multiplatform plugin works with Gradle 6.8.3 or later.

Kotlin

```kotlin
plugins {
    kotlin("multiplatform") version "1.9.0"
}
```

Groovy

```groovy
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}
```

The kotlin-multiplatform plugin configures the project for creating an application or library to work on multiple platforms and prepares it for building on these platforms.

In the file build.gradle(.kts), it creates the kotlin extension at the top level, which includes configuration for targets, source sets, and dependencies.

Targets

A multiplatform project is aimed at multiple platforms that are represented by different targets. A target is part of the build that is responsible for building, testing, and packaging the application for a specific platform, such as macOS, iOS, or Android. See the list of supported platforms.

When you create a multiplatform project, targets are added to the kotlin block in the file build.gradle(.kts).

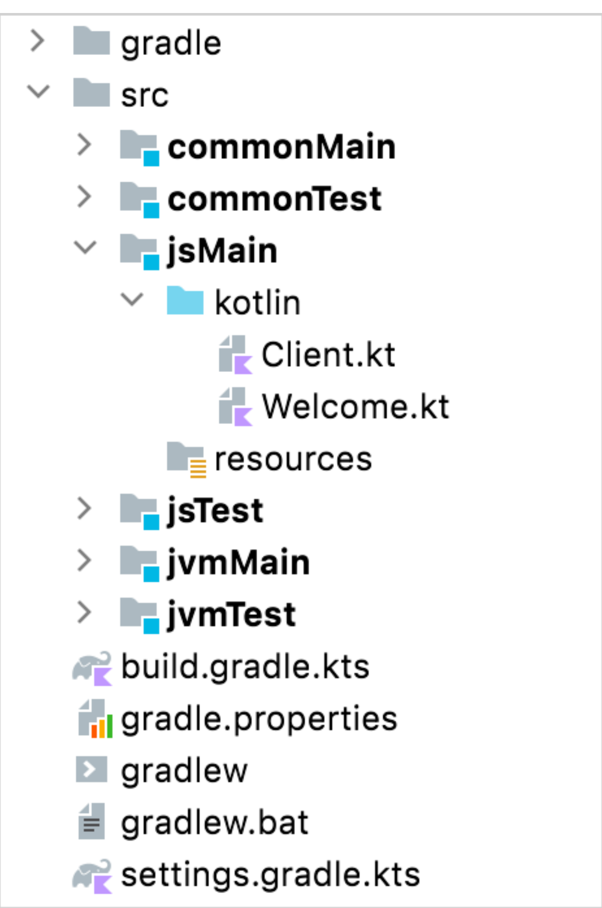
```kotlin
kotlin {
    jvm()
    js {
        browser {}
    }
 }
```

Learn how to set up targets manually.

Source sets

The project includes the directory src with Kotlin source sets, which are collections of Kotlin code files, along with their resources, dependencies, and language settings. A source set can be used in Kotlin compilations for one or more target platforms.

Each source set directory includes Kotlin code files (the kotlin directory) and resources. The Project Wizard creates default source sets for the main and test compilations of the common code and all added targets.



Source sets

Source set names are case-sensitive.

Source sets are added to the sourceSets block of the top-level kotlin block. For example, this is the source sets structure you get when creating a multiplatform library with the IntelliJ IDEA project wizard:

Kotlin

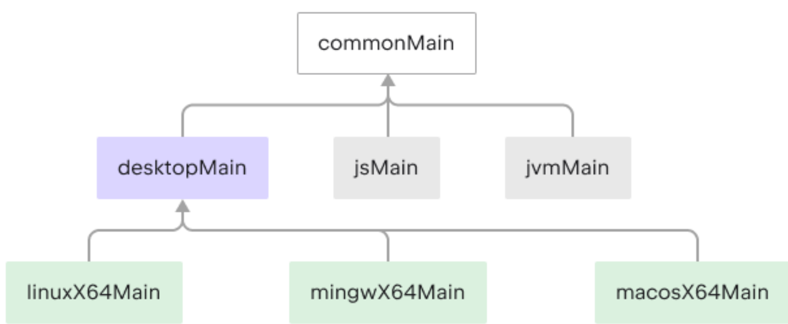
```
kotlin {
    sourceSets {
        val commonMain by getting
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
        val jvmMain by getting
        val jvmTest by getting
        val jsMain by getting
        val jsTest by getting
        val nativeMain by getting
        val nativeTest by getting
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {

        }
        commonTest {
            dependencies {
                implementation kotlin('test')
            }
        }
        jvmMain {

        }
        jvmTest {

        }
        jsMain {

        }
        jsTest {

        }
        nativeMain {

        }
        nativeTest {

        }
    }
}
```

Source sets form a hierarchy, which is used for sharing the common code. In a source set shared among several targets, you can use the platform-specific language features and dependencies that are available for all these targets.

For example, all Kotlin/Native features are available in the desktopMain source set, which targets the Linux (linuxX64), Windows (mingwX64), and macOS (macosX64) platforms.



Hierarchical structure

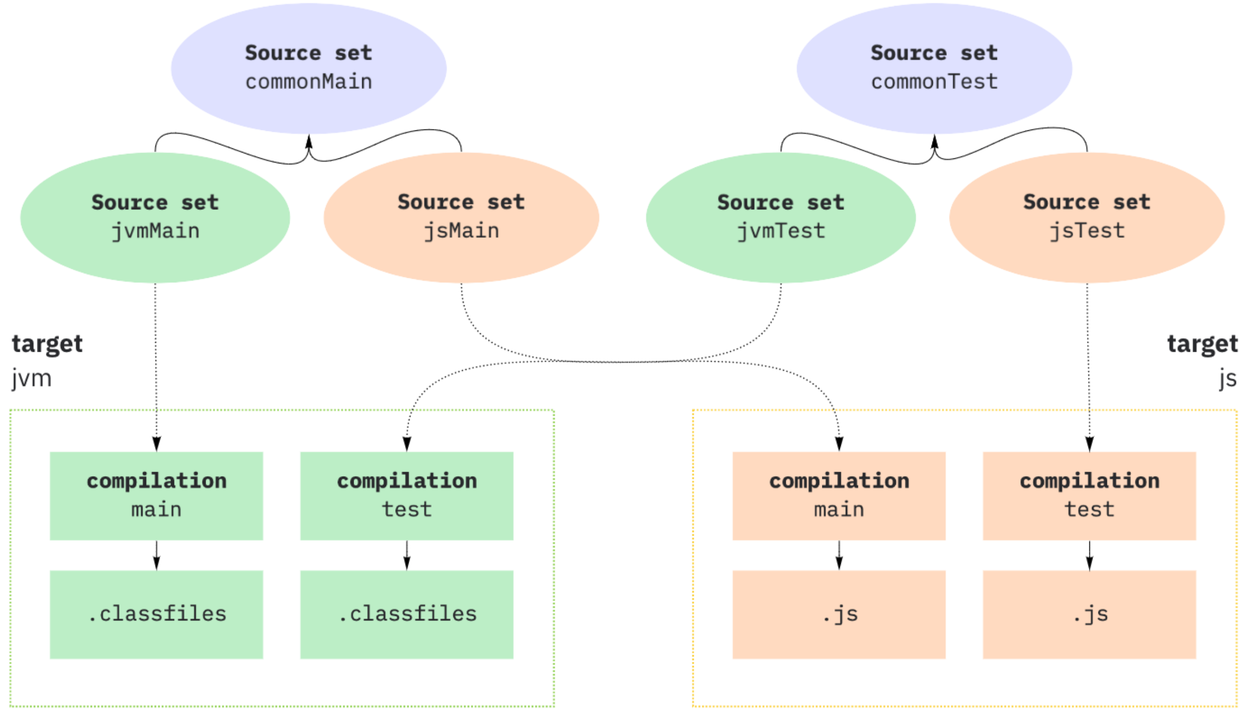
Learn how to build the hierarchy of source sets.

Compilations

Each target can have one or more compilations, for example, for production and test purposes.

For each target, default compilations include:

- main and test compilations for JVM, JS, and Native targets.
- A compilation per Android build variant, for Android targets.



Compilations

Each compilation has a default source set, which contains sources and dependencies specific to that compilation.

Learn how to configure compilations.

Set up targets for Kotlin Multiplatform

You can add targets when creating a project with the Project Wizard. If you need to add a target later, you can do this manually using target presets for supported platforms.

Learn more about additional settings for targets.

```
kotlin {
    jvm() // Create a JVM target with the default name 'jvm'

    linuxX64() {
        /* Specify additional settings for the 'linux' target here */
    }
}
```

Each target can have one or more compilations. In addition to default compilations for test and production purposes, you can create custom compilations.

Distinguish several targets for one platform

You can have several targets for one platform in a multiplatform library. For example, these targets can provide the same API but use different libraries during runtime, such as testing frameworks and logging solutions. Dependencies on such a multiplatform library may fail to resolve because it isn't clear which target to choose.

To solve this, mark the targets on both the library author and consumer sides with a custom attribute, which Gradle uses during dependency resolution.

For example, consider a testing library that supports both JUnit and TestNG in the two targets. The library author needs to add an attribute to both targets as follows:

Kotlin

```kotlin
val testFrameworkAttribute = Attribute.of("com.example.testFramework", String::class.java)

kotlin {
    jvm("junit") {
        attributes.attribute(testFrameworkAttribute, "junit")
    }
    jvm("testng") {
        attributes.attribute(testFrameworkAttribute, "testng")
    }
}
```

Groovy

```groovy
def testFrameworkAttribute = Attribute.of('com.example.testFramework', String)

kotlin {
    jvm('junit') {
        attributes.attribute(testFrameworkAttribute, 'junit')
    }
    jvm('testng') {
        attributes.attribute(testFrameworkAttribute, 'testng')
    }
}
```

The consumer has to add the attribute to a single target where the ambiguity arises.

Build a full-stack web app with Kotlin Multiplatform

This tutorial demonstrates how to build a connected full-stack application with IntelliJ IDEA. You will create a simple JSON API and learn how to use the API from a web app using Kotlin and React.

The application consists of a server part using Kotlin/JVM and a web client using Kotlin/JS. Both parts will be one Kotlin Multiplatform project. Since the whole app will be in Kotlin, you can share libraries and programming paradigms (such as using Coroutines for concurrency) on both the frontend and backend.

Using Kotlin throughout the whole stack also makes it possible to write classes and functions that can be used from both the JVM and JS targets of your application. In this tutorial, you'll primarily utilize this functionality to share a type-safe representation of the data between client and server.

You will also use popular Kotlin multiplatform libraries and frameworks:

- kotlinx.serialization
- kotlinx.coroutines
- The Ktor framework

Serialization and deserialization to and from type-safe objects is delegated to the kotlinx.serialization multiplatform library. This helps you make data communication safe and easy to implement.

The output will be a simple shopping list application that allows you to plan your grocery shopping.

- The user interface will be simple: a list of planned purchases and a field to enter new shopping items.
- If a user clicks on an item in the shopping list, it will be removed.
- The user can also specify a priority level for list entries by adding an exclamation point !. This information will help order the shopping list.

> For this tutorial, you are expected to have an understanding of Kotlin. Some knowledge about basic concepts in React and Kotlin coroutines may help understand some sample code, but it is not strictly required.

Create the project

Clone the project repository from GitHub and open it in IntelliJ IDEA. This template already includes all of the configuration and required dependencies for all of the

project parts: JVM, JS, and the common code.

You don't need to change the Gradle configuration throughout this tutorial. If you want to get right to programming, feel free to move on directly to the next section.

Alternatively, you can get an understanding of the configuration and project setup in the build.gradle.kts file to prepare for other projects. Check out the sections about the Gradle structure below.

Plugins

Like all Kotlin projects targeting more than one platform, your project uses the Kotlin Multiplatform Gradle plugin. It provides a single point of configuration for the application targets (in this case, Kotlin/JVM and Kotlin/JS) and exposes several lifecycle tasks for them.

Additionally, you'll need two more plugins:

- The application plugin runs the server part of the application that uses JVM.
- The serialization plugin provides multiplatform conversions between Kotlin objects and their JSON text representation.

```
plugins {
    kotlin("multiplatform") version "1.9.0"
    application // to run the JVM part
    kotlin("plugin.serialization") version "1.9.0"
}
```

Targets

The target configuration inside the kotlin block is responsible for setting up the platforms you want to support with your project. Configure two targets: jvm (server) and js (client). Here you'll make further adjustments to target configurations.

```
jvm {
    withJava()
}
js {
    browser {
        binaries.executable()
    }
}
```

For more detailed information on targets, see Understand Multiplatform project structure.

Source sets

Kotlin source sets are a collection of Kotlin sources and their resources, dependencies, and language settings that belong to one or more targets. You use them to set up platform-specific and common dependency blocks.

```
sourceSets {
    val commonMain by getting {
        dependencies {
            // ...
        }
    }
    val jvmMain by getting {
        dependencies {
            // ...
        }
    }
    val jsMain by getting {
        dependencies {
            // ...
        }
    }
}
```

Each source set also corresponds to a folder in the src directory. In your project, there are three folders, commonMain, jsMain, and jvmMain, which contain their own resources and kotlin folders.

For detailed information on source sets, see Understand Multiplatform project structure.

Build the backend

Let's begin by writing the server side of the application. The typical API server implements the CRUD operations – create, read, update, and delete. For the simple shopping list, you can focus solely on:

- Creating new entries in the list
- Reading entries using the API
- Deleting entries

To create the backend, you can use the Ktor framework, designed to build asynchronous servers and clients in connected systems. It can be set up quickly and grow as systems become more complex.

You can find more information about Ktor in its documentation.

Run the embedded server

Instantiate a server with Ktor. You need to tell the embedded server that ships with Ktor to use the Netty engine on a port, in this case, 9090.

1. To define the entry point for the app, add the following code to src/jvmMain/kotlin/Server.kt:

```
import io.ktor.http.*
import io.ktor.serialization.kotlinx.json.*
import io.ktor.server.engine.*
import io.ktor.server.netty.*
import io.ktor.server.application.*
import io.ktor.server.plugins.compression.*
import io.ktor.server.plugins.contentnegotiation.*
import io.ktor.server.plugins.cors.routing.*
import io.ktor.server.request.*
import io.ktor.server.response.*
import io.ktor.server.http.content.*
import io.ktor.server.routing.*

fun main() {
    embeddedServer(Netty, 9090) {
        routing {
            get("/hello") {
                call.respondText("Hello, API!")
            }
        }
    }.start(wait = true)
}
```

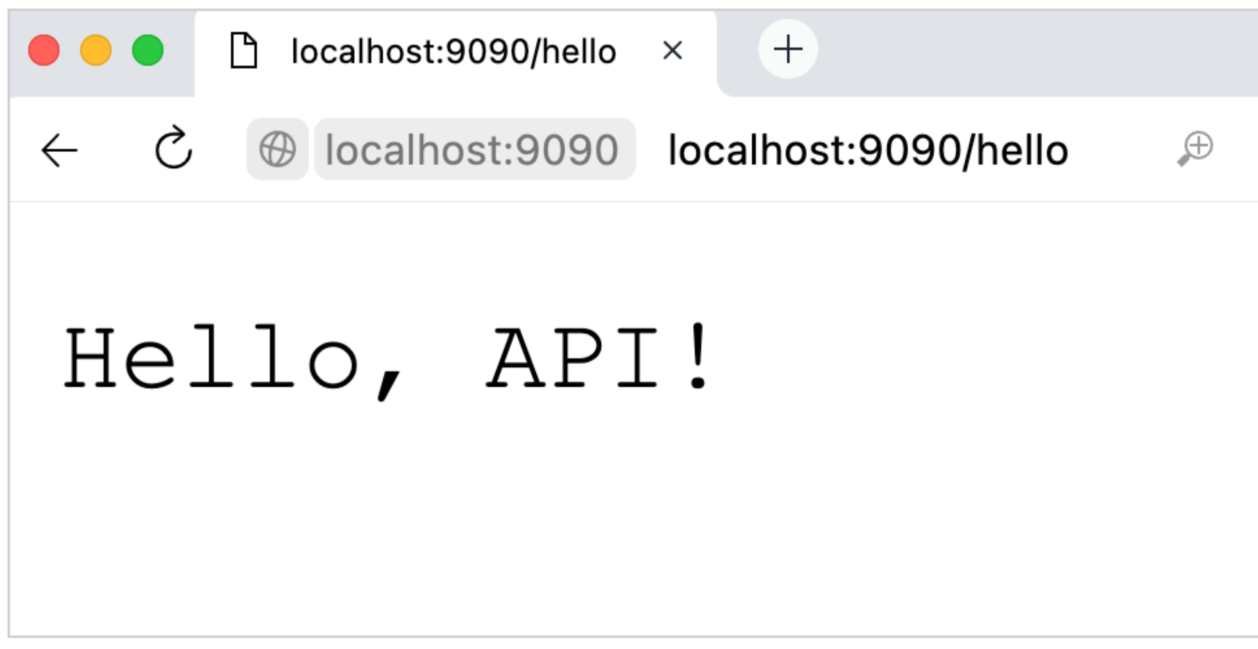
   - The first API endpoint is an HTTP method, get, and the route under which it should be reachable, /hello.
   - All imports that are needed for the rest of the tutorial have already been added.

2. To start the application and see that everything works, execute the Gradle run task. You can use the ./gradlew run command in the terminal or run from the Gradle tool window:

Execute the Gradle run task

3. Once the application has finished compiling and the server has started up, use a web browser to navigate to http://localhost:9090/hello to see the first route in

action:



Hello, API output

Later, like with the endpoint for GET requests to /hello, you'll be able to configure all of the endpoints for the API inside the routing block.

Install Ktor plugins

Before continuing with the application development, install the required plugins for the embedded servers. Ktor uses plugins to enable support for more features in the application like encoding, compression, logging, and authentication.

Add the following lines to the top of the embeddedServer block in src/jvmMain/kotlin/Server.kt:

```
install(ContentNegotiation) {
    json()
}
install(CORS) {
    allowMethod(HttpMethod.Get)
    allowMethod(HttpMethod.Post)
    allowMethod(HttpMethod.Delete)
    anyHost()
}
install(Compression) {
    gzip()
}
routing {
    // ...
}
```

Each call to install adds one feature to the Ktor application:

- ContentNegotiation provides automatic content conversion of requests based on their Content-Type and Accept headers. Together with the json() setting, this enables automatic serialization and deserialization to and from JSON, allowing you to delegate this task to the framework.
- CORS configures Cross-Origin Resource Sharing. CORS is needed to make calls from arbitrary JavaScript clients and helps prevent issues later.
- Compression greatly reduces the amount of data to be sent to the client by gzipping outgoing content when applicable.

Related Gradle configuration for Ktor

The artifacts required to use Ktor are a part of the jvmMain dependencies block in the build.gradle.kts file. This includes the server, logging, and supporting libraries for providing type-safe serialization support through kotlinx.serialization.

```
val jvmMain by getting {
```

```
    dependencies {
        implementation("io.ktor:ktor-serialization:$ktorVersion")
        implementation("io.ktor:ktor-server-content-negotiation:$ktorVersion")
        implementation("io.ktor:ktor-serialization-kotlinx-json:$ktorVersion")
        implementation("io.ktor:ktor-server-cors:$ktorVersion")
        implementation("io.ktor:ktor-server-compression:$ktorVersion")
        implementation("io.ktor:ktor-server-core-jvm:$ktorVersion")
        implementation("io.ktor:ktor-server-netty:$ktorVersion")
        implementation("ch.qos.logback:logback-classic:$logbackVersion")
        implementation("org.litote.kmongo:kmongo-coroutine-serialization:$kmongoVersion")
    }
}
```

kotlinx.serialization and its integration with Ktor also requires a few common artifacts to be present, which you can find in the commonMain source set:

```
val commonMain by getting {
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:$serializationVersion")
        implementation("io.ktor:ktor-client-core:$ktorVersion")
    }
}
```

Create a data model

Thanks to Kotlin Multiplatform, you can define the data model once as a common abstraction and refer to it from both the backend and frontend.

The data model for ShoppingListItem should have:

- A textual description of an item
- A numeric priority for an item
- An identifier

In src/commonMain/, create a kotlin/ShoppingListItem.kt file with the following content:

```
import kotlinx.serialization.Serializable

@Serializable
data class ShoppingListItem(val desc: String, val priority: Int) {
    val id: Int = desc.hashCode()

    companion object {
        const val path = "/shoppingList"
    }
}
```

- The @Serializable annotation comes from the multiplatform kotlinx.serialization library, which allows you to define models directly in common code.
- Once you use this serializable ShoppingListItem class from the JVM and JS platforms, code for each platform will be generated. This code takes care of serialization and deserialization.
- The companion object stores additional information about the model – in this case, the path under which you will be able to access it in the API. By referring to this variable instead of defining routes and requests as strings, you can change the path to model operations. Any changes to the endpoint name only need to be made here - the client and server are adjusted automatically.

> This sample computes a simple id from the hashCode() of its description. In this case, that's enough, but when working with real data, it would be preferable to include tried and tested mechanisms to generate identifiers for your objects – from UUIDs to auto-incrementing IDs backed by the database of your choice.

Add items to store

You can now use the ShoppingListItem model to instantiate some sample items and keep track of any additions or deletions made through the API.

Because there's currently no database, create a MutableList to temporarily store the ShoppingListItems. For that, add the following file-level declaration to src/jvmMain/kotlin/Server.kt:

```
val shoppingList = mutableListOf(
```

```
    ShoppingListItem("Cucumbers 🥒", 1),
    ShoppingListItem("Tomatoes 🍅", 2),
    ShoppingListItem("Orange Juice 🍊", 3)
)
```

The common classes are referred to as any other class in Kotlin – they are shared between all of the targets.

Create routes for the JSON API

Add the routes that support the creation, retrieval, and deletion of ShoppingListItems.

1. Inside src/jvmMain/kotlin/Server.kt, change your routing block to look as follows:

```
routing {
    route(ShoppingListItem.path) {
        get {
            call.respond(shoppingList)
        }
        post {
            shoppingList += call.receive<ShoppingListItem>()
            call.respond(HttpStatusCode.OK)
        }
        delete("/{id}") {
            val id = call.parameters["id"]?.toInt() ?: error("Invalid delete request")
            shoppingList.removeIf { it.id == id }
            call.respond(HttpStatusCode.OK)
        }
    }
}
```

Routes are grouped based on a common path. You don't have to specify the route path as a String. Instead, the path from the ShoppingListItem model is used. The code behaves as follows:

- A get request to the model's path (/shoppingList) responds with the whole shopping list.
- A post request to the model's path (/shoppingList) adds an entry to the shopping list.
- A delete request to the model's path and a provided id (shoppingList/47) removes an entry from the shopping list.

> You can receive objects directly from requests and respond to requests with objects (and even lists of objects) directly. Because you've set up ContentNegotiation with json() support earlier, the objects marked as @Serializable are automatically turned into JSON before being sent (in case of a GET request) or received (in case of a POST request).

2. Check to ensure that everything is working as planned. Restart the application, head to http://localhost:9090/shoppingList, and validate that the data is properly served. You should see the example items in JSON formatting:

localhost:9090/shoppi ×
localhost:9090 localhost:9090/shoppingList

[{"desc":"Cucumbers 🥒","priority":1,"id":2040789031},
{"desc":"Tomatoes 🍅","priority":2,"id":-798383},
{"desc":"Orange Juice 🍊","priority":3,"id":-1137118464}]

Shopping list in JSON formatting

To test the post and delete requests, use an HTTP client that supports .http files. If you're using IntelliJ IDEA Ultimate Edition, you can do this right from the IDE.

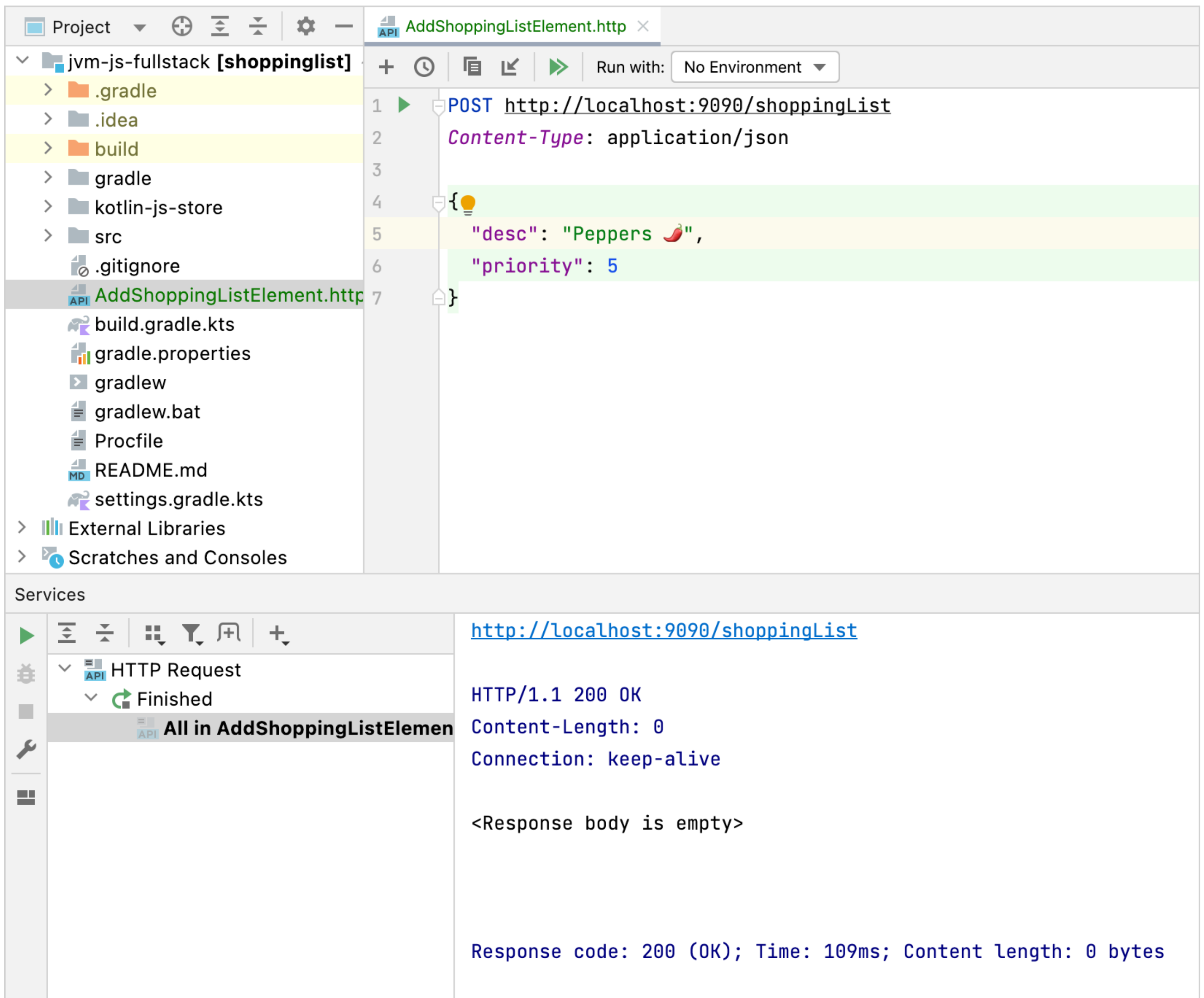
1. In the project root, create a file called AddShoppingListElement.http and add a declaration of the HTTP POST request as follows:

```
POST http://localhost:9090/shoppingList
Content-Type: application/json
```

```
{
  "desc": "Peppers 🌶",
  "priority": 5
}
```

2. With the server running, execute the request using the run button in the gutter.

   If everything goes well, the "run" tool window should show HTTP/1.1 200 OK, and you can visit http://localhost:9090/shoppingList again to validate that the entry has been added properly:



Successful connection to localhost

3. Repeat this process for a file called DeleteShoppingListElement.http, which contains the following:

```
DELETE http://localhost:9090/shoppingList/AN_ID_GOES_HERE
```

   To try this request, replace AN_ID_GOES_HERE with an existing ID.

Now you have a backend that can support all of the necessary operations for a functional shopping list. Move on to building a JavaScript frontend for the application, which will allow users to easily inspect, add, and check off elements from their shopping list.

Set up the frontend

To make your version of the server usable, build a small Kotlin/JS web app that can query the server's API, display them in the form of a list, and allow the user to add and remove elements.

Serve the frontend

Unless explicitly configured otherwise, a Kotlin Multiplatform project just means that you can build the application for each platform, in this case JVM and JavaScript. However, for the application to function properly, you need to have both the backend and the frontend compiled. In fact, you want the backend to also serve all of the assets belonging to the frontend – an HTML page and the corresponding .js file.

In the template project, the adjustments to the Gradle file have already been made. Whenever you run the server with the run Gradle task, the frontend is also built and included in the resulting artifacts. To learn more about how this works, see the Relevant Gradle configuration section.

The template already comes with a boilerplate index.html file in the src/commonMain/resources folder. It has a root node for rendering components and a script tag that includes the application:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Full Stack Shopping List</title>
    </head>
    <body>
        <div id="root"></div>
        <script src="shoppinglist.js"></script>
    </body>
</html>
```

This file is placed in the common resources instead of a jvm source set to make tasks for running the JS application in the browser (jsBrowserDevelopmentRun and jsBrowserProductionRun) accessible to the file as well. It's helpful if you need to run only the browser application without the backend.

While you don't need to make sure that the file is properly available on the server, you still need to instruct Ktor to provide the .html and .js files to the browser when requested.

Relevant Gradle configuration for the frontend

The Gradle configuration for the application contains a snippet that makes the execution and packaging of the server-side JVM application dependent on the build of your frontend application while respecting the settings regarding development and production from the environment variable. It makes sure that whenever a jar file is built from the application, it includes the Kotlin/JS code:

```
// include JS artifacts in any generated JAR
tasks.getByName<Jar>("jvmJar") {
    val taskName = if (project.hasProperty("isProduction")
        || project.gradle.startParameter.taskNames.contains("installDist")
    ) {
        "jsBrowserProductionWebpack"
    } else {
        "jsBrowserDevelopmentWebpack"
    }
    val webpackTask = tasks.getByName<KotlinWebpack>(taskName)
    dependsOn(webpackTask) // make sure JS gets compiled first
    from(File(webpackTask.destinationDirectory, webpackTask.outputFileName)) // bring output file along into the JAR
}
```

The jvmJar task modified here is called by the application plugin, which is responsible for the run task, and the distributions plugin, which is responsible for the installDist task, amongst others. This means that the combined build will work when you run your application, and also when you prepare it for deployment to another target system or cloud platform.

To ensure that the run task properly recognizes the JS artifacts, the classpath is adjusted as follows:

```
tasks.getByName<JavaExec>("run") {
    classpath(tasks.getByName<Jar>("jvmJar")) // so that the JS artifacts generated by `jvmJar` can be found and served
}
```

Serve HTML and JavaScript files from Ktor

For simplicity, the index.html file will be served on the root route / and expose the JavaScript artifact in the root directory.

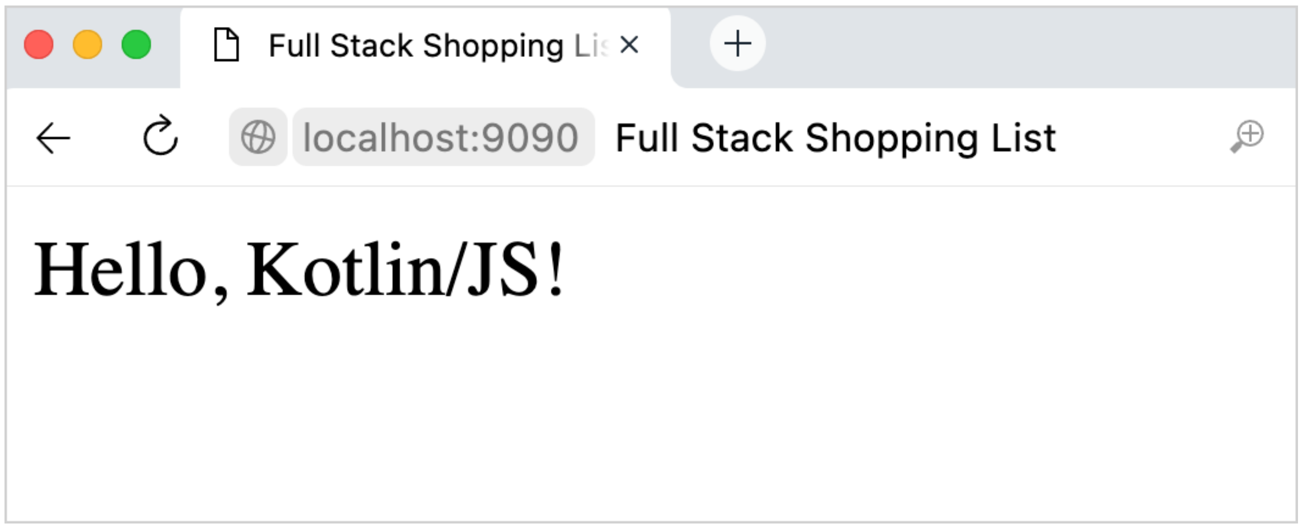
1. In src/jvmMain/kotlin/Server.kt, add the corresponding routes to the routing block:

```
get("/") {
    call.respondText(
        this::class.java.classLoader.getResource("index.html")!!.readText(),
        ContentType.Text.Html
```

```
    )
}
staticResources("/", "static")
route(ShoppingListItem.path) {
    // ...
}
```

2. To confirm that everything went as planned, run the application again with the Gradle run task.

3. Navigate to http://localhost:9090/. You should see a page saying "Hello, Kotlin/JS":



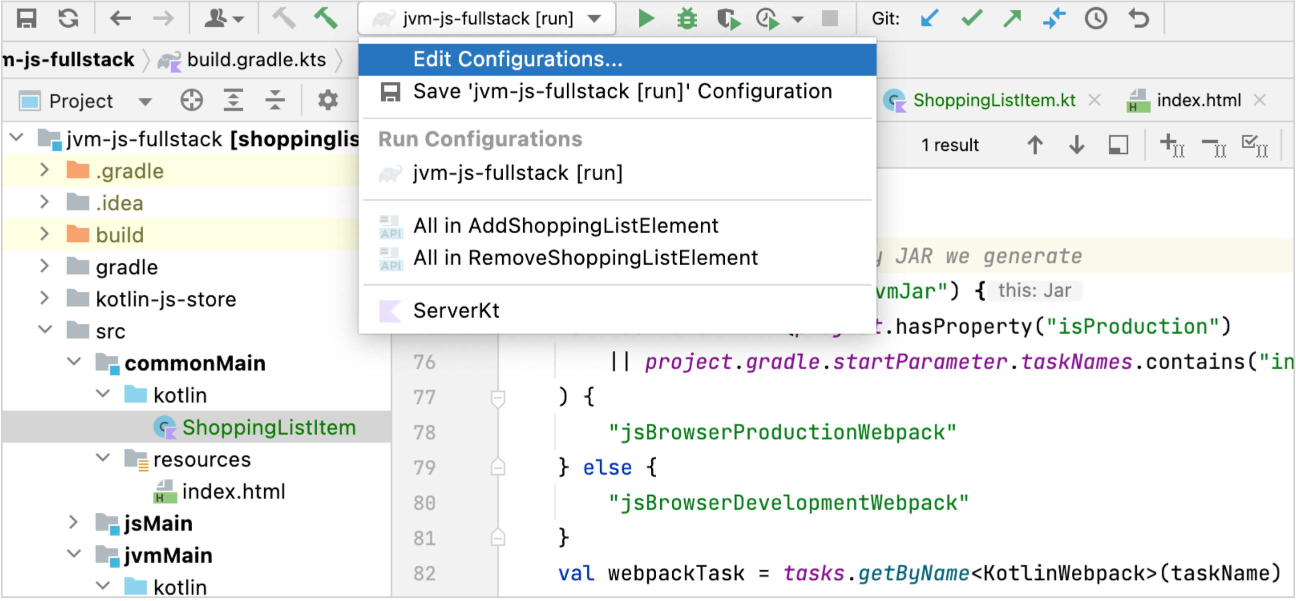
Hello, Kotlin/JS output

Edit configuration

While you are developing, the build system generates development artifacts. This means that no optimizations are applied when the Kotlin code is turned into JavaScript. That makes compile times faster but also results in larger JS files. When you deploy your application to the web, this is something you want to avoid.

To instruct Gradle to generate optimized production assets, set the necessary environment variable. If you are running your application on a deployment system, you can configure it to set this environment variable during the build. If you want to try out production mode locally, you can do it in the terminal or by adding the variable to the run configuration:

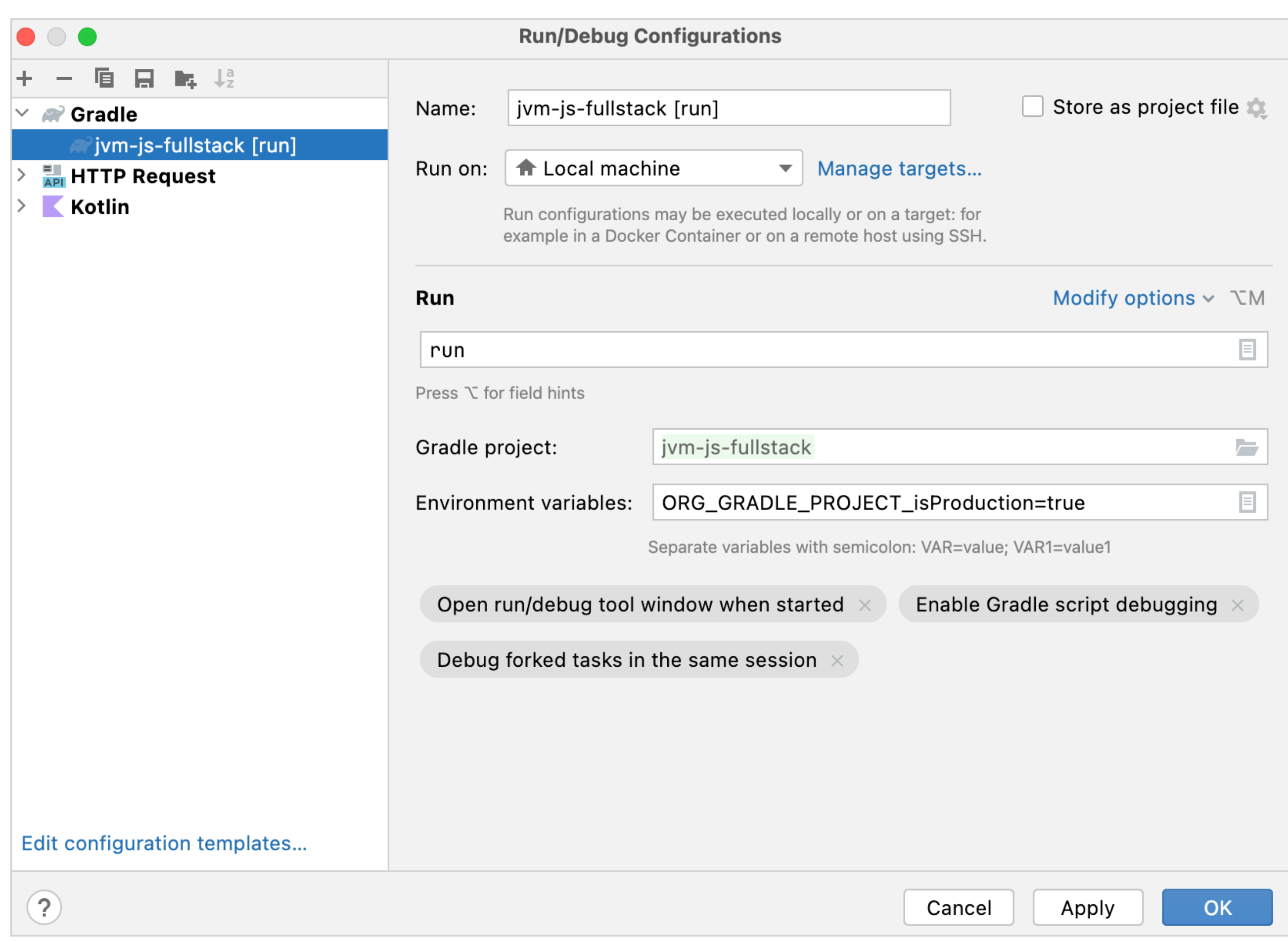
1. In IntelliJ IDEA, select the Edit Configurations action:



Edit run configuration in IntelliJ IDEA

2. In the Run/Debug Configurations menu, set the environment variable:

```
ORG_GRADLE_PROJECT_isProduction=true
```



Set the environment variable

Subsequent builds with this run configuration will perform all available optimizations for the frontend part of the application, including eliminating dead code. They will still be slower than development builds, so it would be good to remove this flag again while you are developing.

Build the frontend

To render and manage user interface elements, use the popular framework React together with the available wrappers for Kotlin. Setting up a full project with React will allow you to re-use it and its configuration as a starting point for more complex multiplatform applications.

For a more in-depth view of typical workflows and how apps are developed with React and Kotlin/JS, see the Build a web application with React and Kotlin/JS tutorial.

Write the API client

To display data, you need to obtain it from the server. For this, build a small API client.

This API client will use the ktor-clients library to send requests to HTTP endpoints. Ktor clients use Kotlin's coroutines to provide non-blocking networking and support plugins like the Ktor server.

In this configuration, the JsonFeature uses kotlinx.serialization to provide a way to create typesafe HTTP requests. It takes care of automatically converting between Kotlin objects and their JSON representation and vice versa.

By leveraging these properties, you can create an API wrapper as a set of suspending functions that either accept or return ShoppingItems. Create a file called

Api.kt and implement them in src/jsMain/kotlin:

```
import io.ktor.http.*
import io.ktor.client.*
import io.ktor.client.call.*
import io.ktor.client.plugins.contentnegotiation.*
import io.ktor.client.request.*
import io.ktor.serialization.kotlinx.json.*

val jsonClient = HttpClient {
    install(ContentNegotiation) {
        json()
    }
}

suspend fun getShoppingList(): List<ShoppingListItem> {
    return jsonClient.get(ShoppingListItem.path).body()
}

suspend fun addShoppingListItem(shoppingListItem: ShoppingListItem) {
    jsonClient.post(ShoppingListItem.path) {
        contentType(ContentType.Application.Json)
        setBody(shoppingListItem)
    }
}

suspend fun deleteShoppingListItem(shoppingListItem: ShoppingListItem) {
    jsonClient.delete(ShoppingListItem.path + "/${shoppingListItem.id}")
}
```

Build the user interface

You've laid the groundwork on the client and have a clean API to access the data provided by the server. Now you can work on displaying the shopping list on the screen in a React application.

Configure an entry point for the application

Instead of rendering a simple "Hello, Kotlin/JS" string, make the application render a functional App component. For that, replace the content inside src/jsMain/kotlin/Main.kt with the following:

```
import web.dom.document
import react.create
import react.dom.client.createRoot

fun main() {
    val container = document.getElementById("root") ?: error("Couldn't find container!")
    createRoot(container).render(App.create())
}
```

Build and render the shopping list

Next, implement the App component. For the shopping list application, it needs to:

- Keep the "local state" of the shopping list to understand which elements to display.
- Load the shopping list elements from the server and set the state accordingly.
- Provide React with instructions on how to render the list.

Based on these requirements, you can implement the App component as follows:

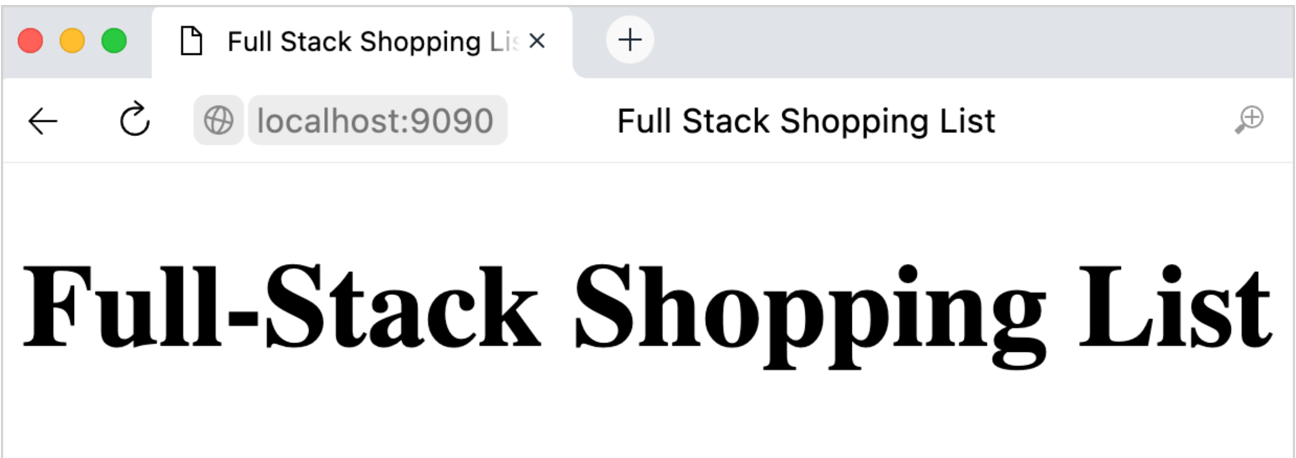
1. Create and fill the src/jsMain/kotlin/App.kt file:

```
import react.*
import kotlinx.coroutines.*
import react.dom.html.ReactHTML.h1
import react.dom.html.ReactHTML.li
import react.dom.html.ReactHTML.ul

private val scope = MainScope()

val App = FC<Props> {
    var shoppingList by useState(emptyList<ShoppingListItem>())
```

```
    useEffectOnce {
        scope.launch {
            shoppingList = getShoppingList()
        }
    }

    h1 {
        +"Full-Stack Shopping List"
    }
    ul {
        shoppingList.sortedByDescending(ShoppingListItem::priority).forEach { item ->
            li {
                key = item.toString()
                +"[${item.priority}] ${item.desc} "
            }
        }
    }
}
```

- Here, the Kotlin DSL is used to define the HTML representation of the application.
- launch is used to obtain the list of ShoppingListItems from the API when the component is first initialized.
- The React hooks useEffectOnce and useState help you use React's functionality concisely. For more information on how React hooks work, check out the official React documentation. To learn more about React with Kotlin/JS, see the Build a web application with React and Kotlin/JS tutorial.

2. Start the application using the Gradle run task.
3. Navigate to http://localhost:9090/ to see the list:



New shopping list rendering

Add an input field component

Next, allow users to add new entries to the shopping list using a text input field. You'll need an input component that provides a callback when users submit their entry to the shopping list to receive input.

1. Create the src/jsMain/kotlin/InputComponent.kt file and fill it with the following definition:

```
import web.html.HTMLFormElement
import react.*
```

```
import web.html.HTMLInputElement
import react.dom.events.ChangeEventHandler
import react.dom.events.FormEventHandler
import web.html.InputType
import react.dom.html.ReactHTML.form
import react.dom.html.ReactHTML.input

external interface InputProps : Props {
    var onSubmit: (String) -> Unit
}

val inputComponent = FC<InputProps> { props ->
    val (text, setText) = useState("")

    val submitHandler: FormEventHandler<HTMLFormElement> = {
        it.preventDefault()
        setText("")
        props.onSubmit(text)
    }

    val changeHandler: ChangeEventHandler<HTMLInputElement> = {
        setText(it.target.value)
    }

    form {
        onSubmit = submitHandler
        input {
            type = InputType.text
            onChange = changeHandler
            value = text
        }
    }
}
```

The inputComponent keeps track of its internal state (what the user has typed so far) and exposes an onSubmit handler that gets called when the user submits the form (usually by pressing the Enter key).

2. To use this inputComponent from the application, add the following snippet to src/jsMain/kotlin/App.kt at the bottom of the FC block (after the closing brace for the ul element):

```
inputComponent {
    onSubmit = { input ->
        val cartItem = ShoppingListItem(input.replace("!", ""), input.count { it == '!' })
        scope.launch {
            addShoppingListItem(cartItem)
            shoppingList = getShoppingList()
        }
    }
}
```

- When users submit text, a new ShoppingListItem is created. Its priority is set to be the number of exclamation points in the input, and its description is the input with all exclamation points removed. This turns Peaches!! 🍑 into a ShoppingListItem(desc="Peaches 🍑", priority=2).
- The generated ShoppingListItem gets sent to the server with the client you've built before.
- Then, the UI is updated by obtaining the new list of ShoppingListItems from the server, updating the application state, and letting React re-render the contents.

Implement item removal

Add the ability to remove the finished items from the list so that it doesn't get too long. You can modify an existing list rather than adding another UI element (like a "delete" button). When users click one of the items in the list, the app deletes it.

To achieve this, pass a corresponding handler to onClick of the list elements:

1. In src/jsMain/kotlin/App.kt, update the li block (inside the ul block):

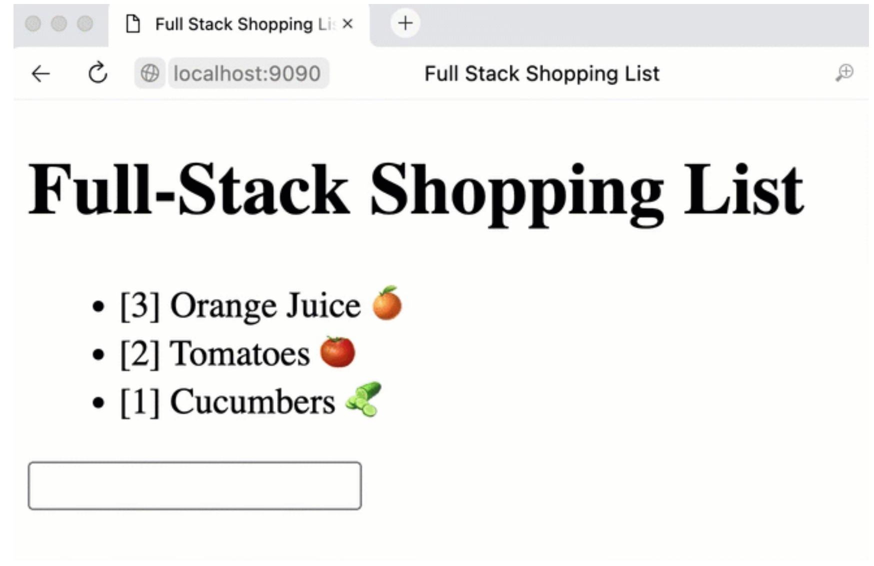
```
li {
    key = item.toString()
    onClick = {
        scope.launch {
            deleteShoppingListItem(item)
            shoppingList = getShoppingList()
        }
    }
```

```
    +"[${item.priority}] ${item.desc} "
}
```

The API client is invoked along with the element that should be removed. The server updates the shopping list, which re-renders the user interface.

2. Start the application using the Gradle run task.

3. Navigate to http://localhost:9090/, and try adding and removing elements from the list:



Final shopping list

Include a database to store data

Currently, the application doesn't save data, meaning that the shopping list vanishes when you terminate the server process. To fix that, use the MongoDB database to store and retrieve shopping list items even when the server shuts down.

MongoDB is simple, fast to set up, has library support for Kotlin, and provides simple, NoSQL document storage, which is more than enough for a basic application. You are free to equip your application with a different mechanism for data storage.

To provide all of the functionality used in this section, you'll need to include several libraries from the Kotlin and JavaScript (npm) ecosystems. See the jsMain dependency block in the build.gradle.kts file with the full setup.

Set up MongoDB

Install MongoDB Community Edition on your local machine from the official MongoDB website. Alternatively, you can use a containerization tool like podman to run a containerized instance of MongoDB.

After installation, ensure that you are running the mongodb-community service for the rest of the tutorial. You'll use it to store and retrieve list entries.

Include KMongo in the process

KMongo is a community-created Kotlin framework that makes it easy to work with MongoDB from Kotlin/JVM code. It also works nicely with kotlinx.serialization, which is used to facilitate communication between client and server.

By making the code use an external database, you no longer need to keep a collection of shoppingListItems on the server. Instead, set up a database client and obtain a database and a collection from it.

1. Inside src/jvmMain/kotlin/Server.kt, remove the declaration for shoppingList and add the following three top-level variables:

```
val client = KMongo.createClient().coroutine
val database = client.getDatabase("shoppingList")
val collection = database.getCollection<ShoppingListItem>()
```

2. In src/jvmMain/kotlin/Server.kt, replace definitions for the GET, POST, and DELETE routes to a ShoppingListItem to make use of the available collection

operations:

```
get {
    call.respond(collection.find().toList())
}
post {
    collection.insertOne(call.receive<ShoppingListItem>())
    call.respond(HttpStatusCode.OK)
}
delete("/{id}") {
    val id = call.parameters["id"]?.toInt() ?: error("Invalid delete request")
    collection.deleteOne(ShoppingListItem::id eq id)
    call.respond(HttpStatusCode.OK)
}
```

In the DELETE request, KMongo's type-safe queries are used to obtain and remove the correct ShoppingListItem from the database.

3. Start the server using the run task, and navigate to http://localhost:9090/. On the first start, you'll be greeted by an empty shopping list as is expected when querying an empty database.

4. Add some items to your shopping list. The server will save them to the database.

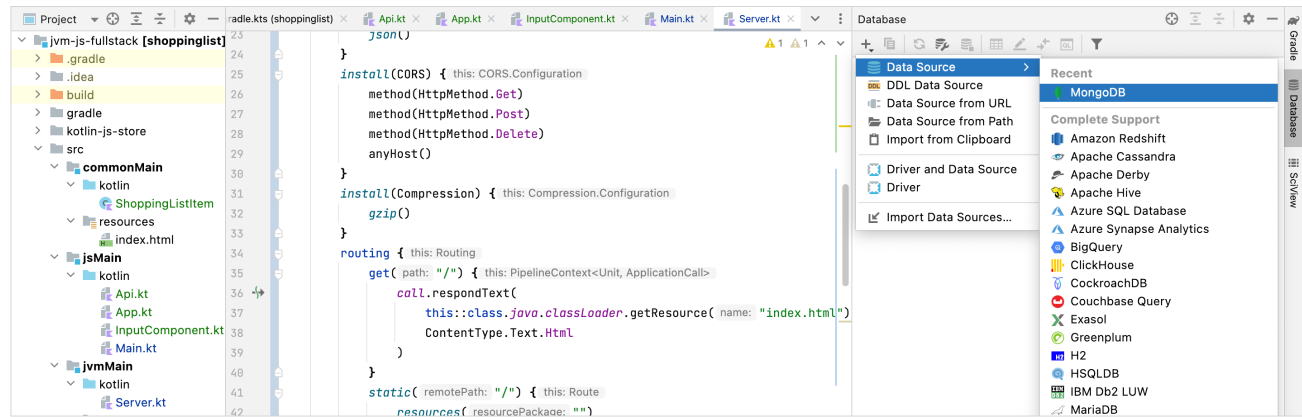
5. To check this, restart the server and reload the page.

Inspect MongoDB

To see what kind of information is actually saved in the database, you can inspect the database using external tools.

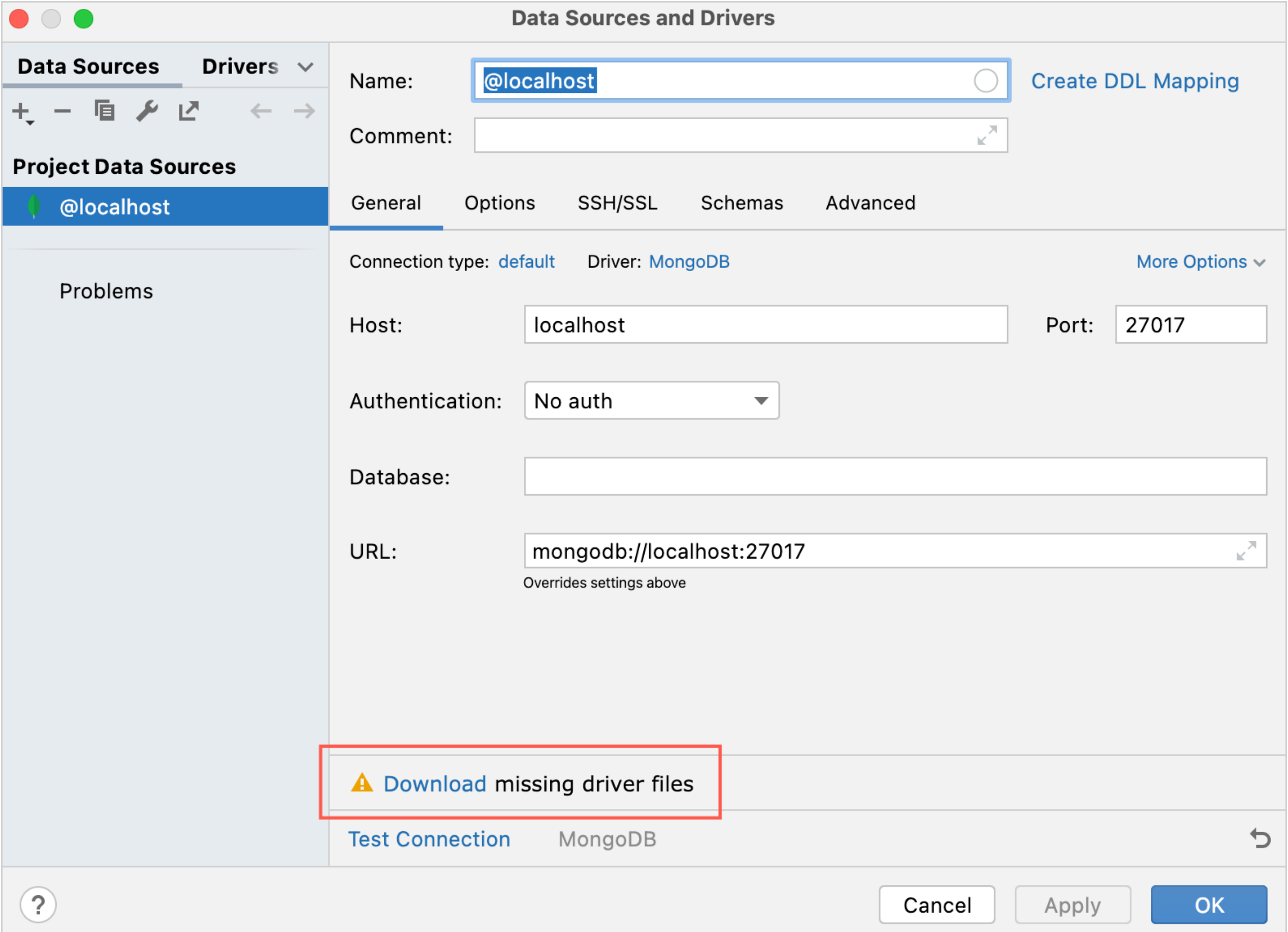
If you have IntelliJ IDEA Ultimate Edition or DataGrip, you can inspect the database contents with these tools. Alternatively, you can use the mongosh command-line client.

1. To connect to the local MongoDB instance, in IntelliJ IDEA Ultimate or DataGrip, go to the Database tab and select + | Data Source | MongoDB:



Create a MongoDB data source

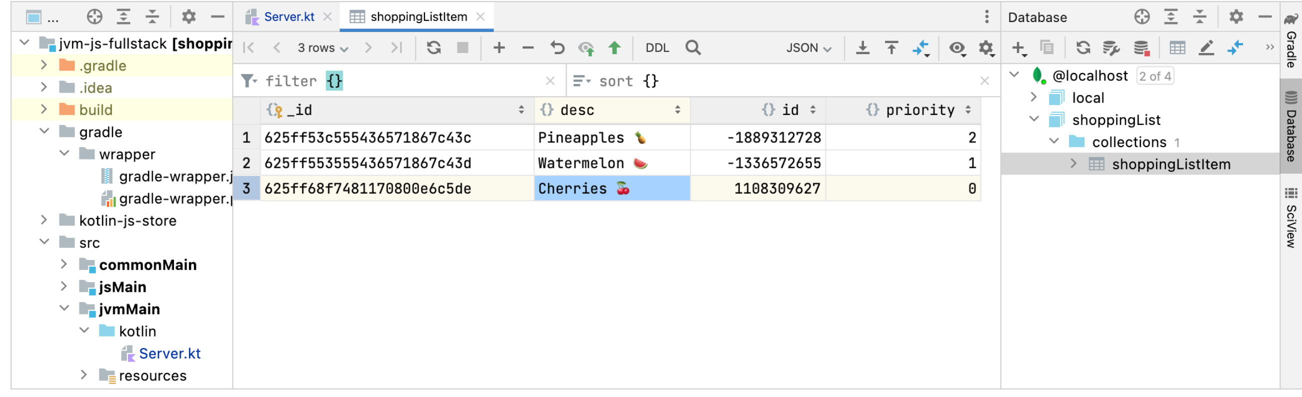
2. If it's your first time connecting to a MongoDB database this way, you might be prompted to download missing drivers:



Download missing drivers for MongoDB

3. When working with a local MongoDB installation that uses the default settings, no adjustments to the configuration are necessary. You can test the connection with the Test Connection button, which should output the MongoDB version and some additional information.

4. Click OK. Now you can use the Database window to navigate to your collection and look at everything stored in it:



Use the Database tool for collection analysis

Relevant Gradle configuration for Kmongo

Kmongo is added with a single dependency to the project, a specific version that includes coroutine and serialization support out of the box:

```
val jvmMain by getting {
    dependencies {
        // ...
```

```
        implementation("org.litote.kmongo:kmongo-coroutine-serialization:$kmongoVersion")
    }
}
```

Deploy to the cloud

Instead of opening your app on localhost, you can bring it onto the web by deploying it to the cloud.

To get the application running on managed infrastructure (such as cloud providers), you need to integrate it with the environment variables provided by the selected platform and add any required configurations to the project. Specifically, pass the application port and MongoDB connection string.

> During application deployment, you might need to change the firewall rules to allow the application to access the database. For more details, see the MongoDB documentation.

Specify the PORT variable

On managed platforms, the port on which the application should run is often determined externally and exposed through the PORT environment variable. If present, you can respect this setting by configuring embeddedServer in src/jvmMain/kotlin/Server.kt:

```
fun main() {
    val port = System.getenv("PORT")?.toInt() ?: 9090
    embeddedServer(Netty, port) {
        // ...
    }
}
```

Ktor also supports configuration files that can respect environment variables. To learn more about how to use them, check out the official documentation.

Specify the MONGODB_URI variable

Managed platforms often expose connection strings through environment variables – for MongoDB, this might be the MONGODB_URI string, which needs to be used by the client to connect to the database. Depending on the specific MongoDB instance you're trying to connect to, you might need to append the retryWrites=false parameter to the connection string.

To properly satisfy these requirements, instantiate the client and database variables in src/jvmMain/kotlin/Server.kt:

```
val connectionString: ConnectionString? = System.getenv("MONGODB_URI")?.let {
    ConnectionString("$it?retryWrites=false")
}

val client =
    if (connectionString != null) KMongo.createClient(connectionString).coroutine else KMongo.createClient().coroutine
val database = client.getDatabase(connectionString?.database ?: "shoppingList")
```

This ensures that the client is created based on this information whenever the environment variables are set. Otherwise (for instance, on localhost), the database connection is instantiated as before.

Create the Procfile

Managed cloud platforms like Heroku or PaaS implementations like Dokku also handle the lifecycle of your application. To do so, they require an "entry point" definition. These two platforms use a file called Procfile that you have in the project root directory. It points to the output generated by the stage task (which is included in the Gradle template already):

```
web: ./build/install/shoppingList/bin/shoppingList
```

Turn on production mode

To turn on a compilation with optimizations for the JavaScript assets, pass another flag to the build process. In the Run/Debug Configurations menu, set the environment variable ORG_GRADLE_PROJECT_isProduction to true. You can set this environment variable when you deploy the application to the target environment.

You can find the finished application on GitHub on the final branch.

Relevant Gradle configuration

The stage task is an alias for installDist:

```
// Alias "installDist" as "stage" (for cloud providers)
tasks.create("stage") {
    dependsOn(tasks.getByName("installDist"))
}

// only necessary until https://youtrack.jetbrains.com/issue/KT-37964 is resolved
distributions {
    main {
        contents {
            from("$buildDir/libs") {
                rename("${rootProject.name}-jvm", rootProject.name)
                into("lib")
            }
        }
    }
}
```

What's next

Add more features

See how your application can be expanded and improved:

- Improve the design. You could make use of styled-components, one of the libraries that have Kotlin wrappers provided. If you want to see styled-components in action, look at the Build a web application with React and Kotlin/JS tutorial.
- Add crossing out list items. For now, list items just vanish with no record of them existing. Instead of deleting an element, use crossing out.
- Implement editing. So far, an entry in the shopping list can't be edited. Consider adding an edit button.

Join the community and get help

You can join the official Kotlin Slack channels, #ktor, #javascript, and others to get help with Kotlin related problems from the community.

Learn more about Kotlin/JS

You can find additional learning materials targeting Kotlin/JS: Set up a Kotlin/JS project and Run Kotlin/JS.

Learn more about Ktor

For in-depth information about the Ktor framework, including demo projects, check out ktor.io.

If you run into trouble, check out the Ktor issue tracker on YouTrack – and if you can't find your problem, don't hesitate to file a new issue.

Learn more about Kotlin Multiplatform

Learn more about how multiplatform code works in Kotlin.

Create and publish a multiplatform library – tutorial

In this tutorial, you will learn how to create a multiplatform library for JVM, JS, and Native platforms, write common tests for all platforms, and publish the library to a local Maven repository.

This library converts raw data – strings and byte arrays – to the Base64 format. It can be used on Kotlin/JVM, Kotlin/JS, and any available Kotlin/Native platform.

You will use different ways to implement the conversion to the Base64 format on different platforms:

* For JVM – the java.util.Base64 class.
* For JS – the btoa() function.
* For Kotlin/Native – your own implementation.

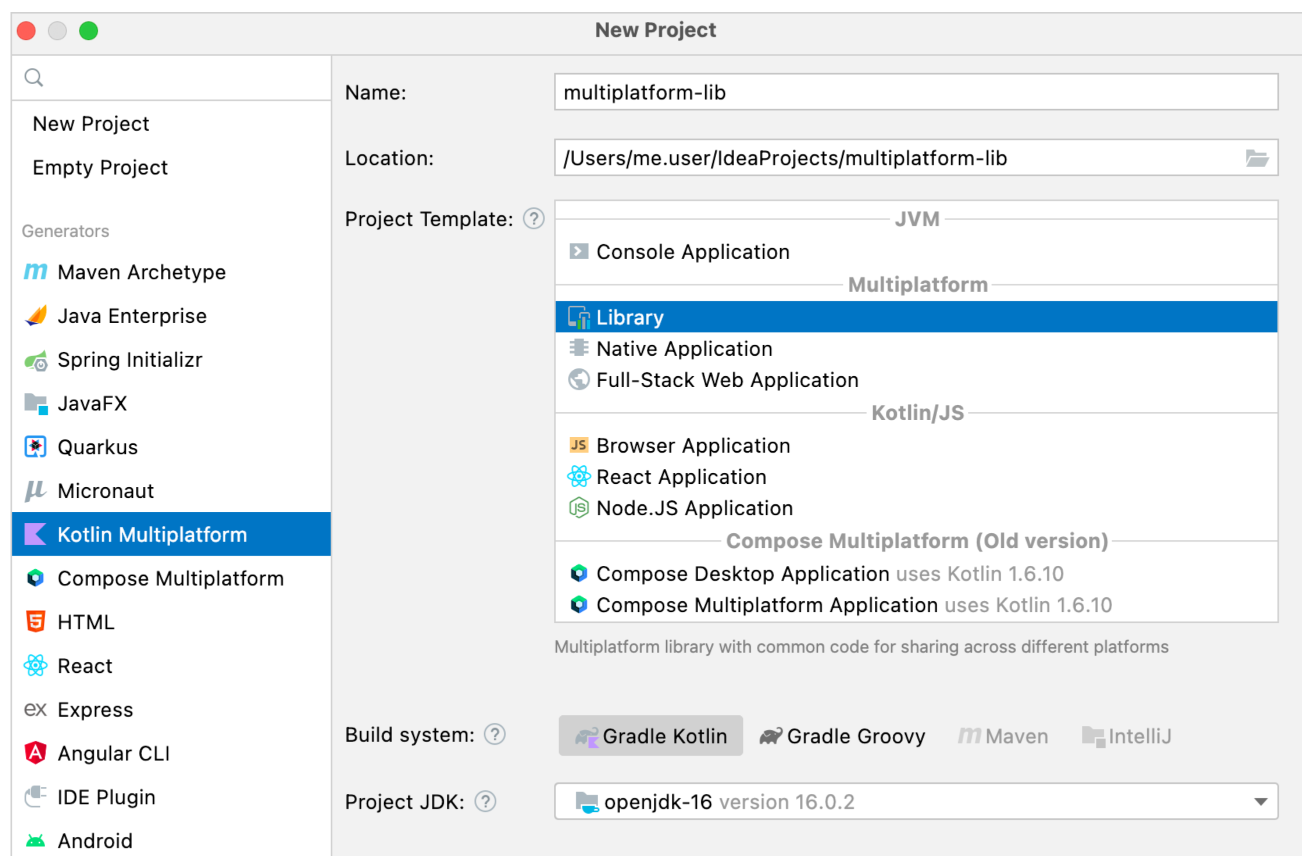
You will also test your code using common tests, and then publish the library to your local Maven repository.

Set up the environment

You can complete this tutorial on any operating system. Download and install the latest version of IntelliJ IDEA with the latest Kotlin plugin.

Create a project

1. In IntelliJ IDEA, select File | New | Project.
2. In the left-hand panel, select Kotlin Multiplatform.
3. Enter a project name, then in the Multiplatform section select Library as the project template.



Select a project template

By default, your project will use Gradle with Kotlin DSL as the build system.

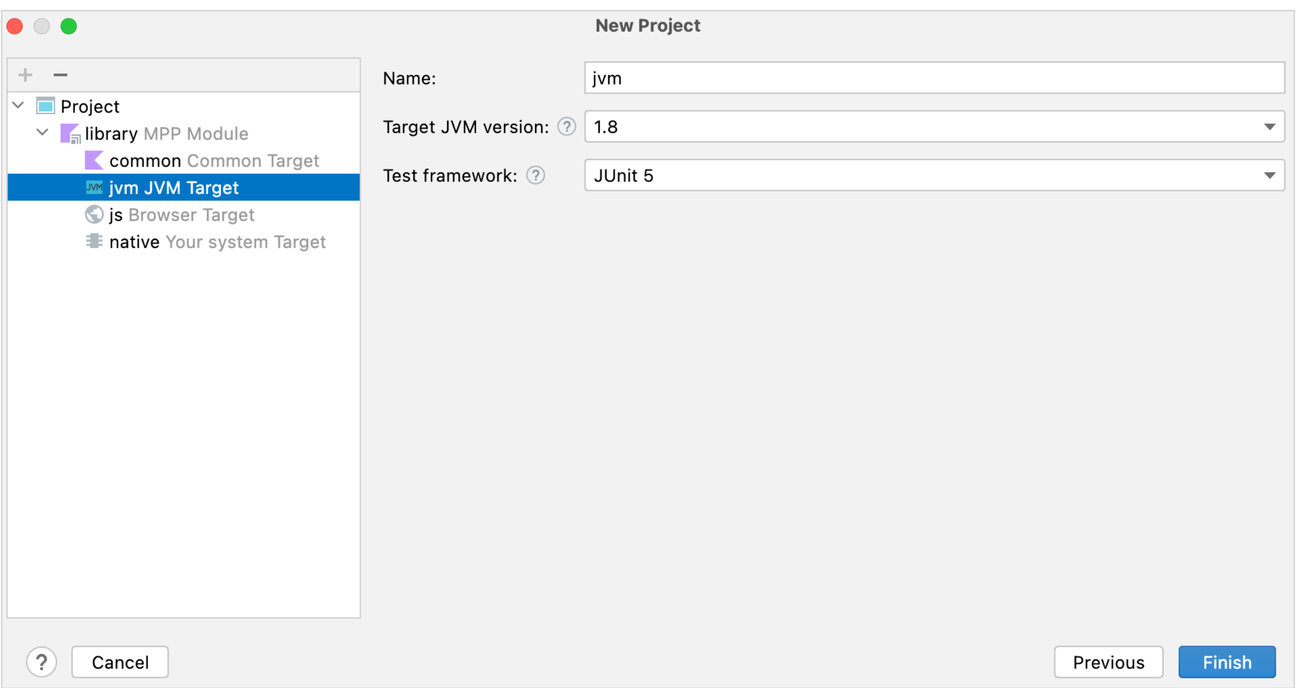
4. Specify the JDK, which is required for developing Kotlin projects.
5. Click Next and then Finish.

Further project configuration

For more complex projects, you might need to add more modules and targets:

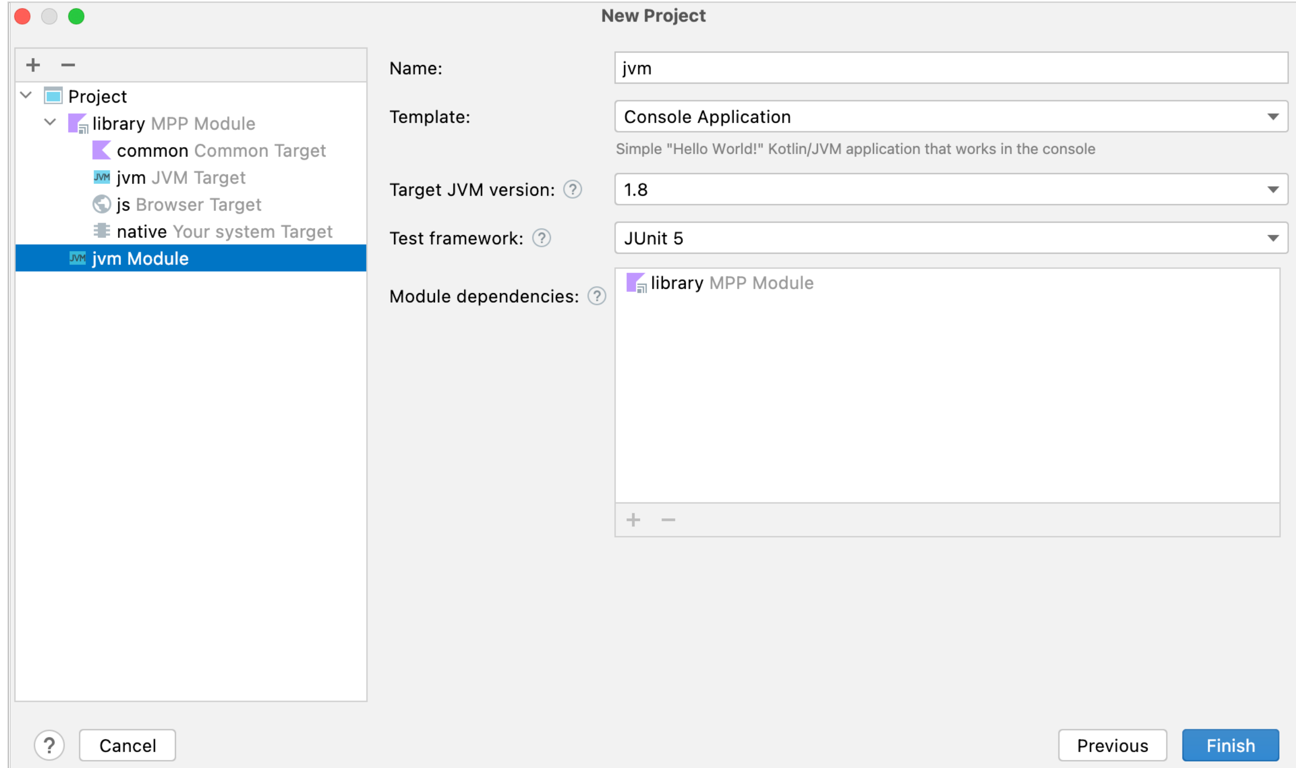
* To add modules, select Project and click the + icon. Choose the module type.
* To add target platforms, select library and click the + icon. Choose the target.

* Configure target settings, such as the target JVM version and test framework.



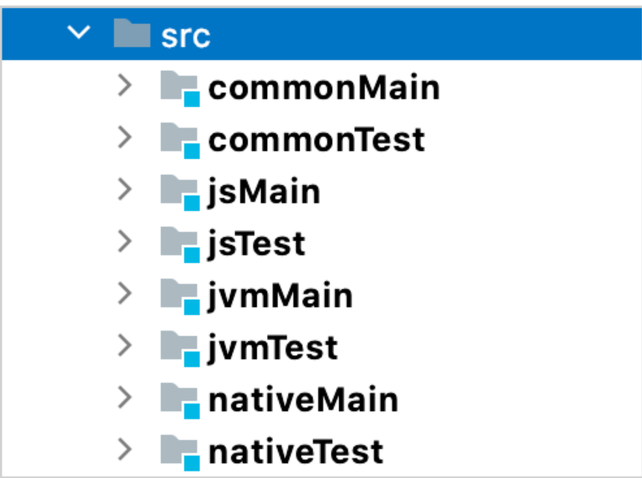
Configure the project

* If necessary, specify dependencies between modules:
  * Multiplatform and Android modules
  * Multiplatform and iOS modules
  * JVM modules



Configure the project

The wizard will create a sample multiplatform library with the following structure:



Multiplatform library structure

Write cross-platform code

Define the classes and interfaces you are going to implement in the common code.

1. In the commonMain/kotlin directory, create the org.jetbrains.base64 package.

2. Create the Base64.kt file in the new package.

3. Define the Base64Encoder interface that converts bytes to the Base64 format:

```
package org.jetbrains.base64

interface Base64Encoder {
    fun encode(src: ByteArray): ByteArray
}
```

4. Define the Base64Factory object to provide an instance of the Base64Encoder interface to the common code:

```
expect object Base64Factory {
    fun createEncoder(): Base64Encoder
}
```

The factory object is marked with the expect keyword in the cross-platform code. For each platform, you should provide an actual implementation of the Base64Factory object with the platform-specific encoder. Learn more about platform-specific implementations.

Provide platform-specific implementations

Now you will create the actual implementations of the Base64Factory object for each platform:

- JVM
- JS
- Native

JVM

1. In the jvmMain/kotlin directory, create the org.jetbrains.base64 package.

2. Create the Base64.kt file in the new package.

3. Provide a simple implementation of the Base64Factory object that delegates to the java.util.Base64 class:

> IDEA inspections help create actual implementations for an expect declaration.

```kotlin
package org.jetbrains.base64
import java.util.*

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = JvmBase64Encoder
}

object JvmBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray = Base64.getEncoder().encode(src)
}
```

Pretty simple, right? You've provided a platform-specific implementation by using a straightforward delegation to a third-party implementation.

JS

The JS implementation will be very similar to the JVM one.

1. In the jsMain/kotlin directory, create the org.jetbrains.base64 package.
2. Create the Base64.kt file in the new package.
3. Provide a simple implementation of the Base64Factory object that delegates to the btoa() function.

```kotlin
package org.jetbrains.base64

import kotlinx.browser.window

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = JsBase64Encoder
}

object JsBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray {
        val string = src.decodeToString()
        val encodedString = window.btoa(string)
        return encodedString.encodeToByteArray()
    }
}
```

Native

Unfortunately, there is no third-party implementation available for all Kotlin/Native targets, so you need to write it yourself.

1. In the nativeMain/kotlin directory, create the org.jetbrains.base64 package.
2. Create the Base64.kt file in the new package.
3. Provide your own implementation for the Base64Factory object:

```kotlin
package org.jetbrains.base64

private val BASE64_ALPHABET: String = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
private val BASE64_MASK: Byte = 0x3f
private val BASE64_PAD: Char = '='
private val BASE64_INVERSE_ALPHABET = IntArray(256) {
    BASE64_ALPHABET.indexOf(it.toChar())
}

private fun Int.toBase64(): Char = BASE64_ALPHABET[this]

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = NativeBase64Encoder
}

object NativeBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray {
```

```
        fun ByteArray.getOrZero(index: Int): Int = if (index >= size) 0 else get(index).toInt()
        // 4n / 3 is expected Base64 payload
        val result = ArrayList<Byte>(4 * src.size / 3)
        var index = 0
        while (index < src.size) {
            val symbolsLeft = src.size - index
            val padSize = if (symbolsLeft >= 3) 0 else (3 - symbolsLeft) * 8 / 6
            val chunk = (src.getOrZero(index) shl 16) or (src.getOrZero(index + 1) shl 8) or src.getOrZero(index + 2)
            index += 3

            for (i in 3 downTo padSize) {
                val char = (chunk shr (6 * i)) and BASE64_MASK.toInt()
                result.add(char.toBase64().code.toByte())
            }
            // Fill the pad with '='
            repeat(padSize) { result.add(BASE64_PAD.code.toByte()) }
        }

        return result.toByteArray()
    }
}
```

Test your library

Now when you have actual implementations of the Base64Factory object for all platforms, it's time to test your multiplatform library.

To save time on testing, you can write common tests that will be executed on all platforms instead of testing each platform separately.

Prerequisites

Before writing tests, add the encodeToString method with the default implementation to the Base64Encoder interface, which is defined in commonMain/kotlin/org/jetbrains/base64/Base64.kt. This implementation converts byte arrays to strings, which are much easier to test.

```
interface Base64Encoder {
    fun encode(src: ByteArray): ByteArray

    fun encodeToString(src: ByteArray): String {
        val encoded = encode(src)
        return buildString(encoded.size) {
            encoded.forEach { append(it.toInt().toChar()) }
        }
    }
}
```

You can also provide a more efficient implementation of this method for a specific platform, for example, for JVM in jvmMain/kotlin/org/jetbrains/base64/Base64.kt:

```
object JvmBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray = Base64.getEncoder().encode(src)
    override fun encodeToString(src: ByteArray): String = Base64.getEncoder().encodeToString(src)
}
```

One of the benefits of a multiplatform library is having a default implementation with optional platform-specific overrides.

Write common tests

Now you have a string-based API that you can cover with basic tests.

1. In the commonTest/kotlin directory, create the org.jetbrains.base64 package.
2. Create the Base64Test.kt file in the new package.
3. Add tests to this file:

```
package org.jetbrains.base64

import kotlin.test.Test
import kotlin.test.assertEquals

class Base64Test {
    @Test
```

```kotlin
    fun testEncodeToString() {
        checkEncodeToString("Kotlin is awesome", "S290bGluIGlzIGF3ZXNvbWU=")
    }

    @Test
    fun testPaddedStrings() {
        checkEncodeToString("", "")
        checkEncodeToString("1", "MQ==")
        checkEncodeToString("22", "MjI=")
        checkEncodeToString("333", "MzMz")
        checkEncodeToString("4444", "NDQ0NA==")
    }

    private fun checkEncodeToString(input: String, expectedOutput: String) {
        assertEquals(expectedOutput, Base64Factory.createEncoder().encodeToString(input.asciiToByteArray()))
    }

    private fun String.asciiToByteArray() = ByteArray(length) {
        get(it).code.toByte()
    }
}
```

4. In the Terminal, execute the check Gradle task:

```
./gradlew check
```

> You can also run the check Gradle task by double-clicking it in the list of Gradle tasks.

The tests will run on all platforms (JVM, JS, and Native).

Add platform-specific tests

You can also add tests that will be run only for a specific platform. For example, you can add UTF-16 tests on JVM:

1. In the jvmTest/kotlin directory, create the org.jetbrains.base64 package.

2. Create the Base64Test.kt file in the new package.

3. Add tests to this file:

```kotlin
package org.jetbrains.base64

import kotlin.test.Test
import kotlin.test.assertEquals

class Base64JvmTest {
    @Test
    fun testNonAsciiString() {
        val utf8String = "Gödel"
        val actual = Base64Factory.createEncoder().encodeToString(utf8String.toByteArray())
        assertEquals("R8O2ZGVs", actual)
    }
}
```

This test will automatically run on the JVM platform in addition to the common tests.

Publish your library to the local Maven repository

Your multiplatform library is ready for publishing so that you can use it in other projects.

To publish your library, use the maven-publish Gradle plugin.

1. In the build.gradle.kts file, apply the maven-publish plugin and specify the group and version of your library:

```kotlin
plugins {
    kotlin("multiplatform") version "1.9.0"
    id("maven-publish")
}
```

```
group = "org.jetbrains.base64"
version = "1.0.0"
```

2. In the Terminal, run the publishToMavenLocal Gradle task to publish your library to your local Maven repository:

```
./gradlew publishToMavenLocal
```

> You can also run the publishToMavenLocal Gradle task by double-clicking it in the list of Gradle tasks.

Your library will be published to the local Maven repository.

Publish your library to the external Maven Central repository

You can go public and release your multiplatform library to Maven Central, a remote repository where maven artifacts are stored and managed. This way, other developers will be able to find it and add as a dependency to their projects.

Register a Sonatype account and generate GPG keys

If this is your first library, or you used the sunset Bintray to do this before, you need first to register a Sonatype account.

You can use the GetStream article to create and set up your account. The Registering a Sonatype account section describes how to:

1. Register a Sonatype Jira account.
2. Create a new issue. You can use our issue as an example.
3. Verify your domain ownership corresponding to the group ID you want to use to publish your artifacts.

Then, since artifacts published on Maven Central have to be signed, follow the Generating a GPG key pair section to:

1. Generate a GPG key pair for signing your artifacts.
2. Publish your public key.
3. Export your private key.

When the Maven repository and signing keys for your library are ready, you can move on and set up your build to upload the library artifacts to a staging repository and then release them.

Set up publication

Now you need to instruct Gradle how to publish the library. Most of the work is already done by the maven-publish and Kotlin Gradle plugins, all the required publications are created automatically. You already know the result when the library is published to a local Maven repository. To publish it to Maven Central, you need to take additional steps:

1. Configure the public Maven repository URL and credentials.
2. Provide a description and javadocs for all library components.
3. Sign publications.

You can handle all these tasks with Gradle scripts. Let's extract all the publication-related logic from the library module build.script, so you can easily reuse it for other modules in the future.

The most idiomatic and flexible way to do that is to use Gradle's precompiled script plugins. All the build logic will be provided as a precompiled script plugin and could be applied by plugin ID to every module of our library.

To implement this, move the publication logic to a separate Gradle project:

1. Add a new Gradle project inside your library root project. For that, create a new folder named convention-plugins and create a new file build.gradle.kts inside of it.
2. Place the following code into the new build.gradle.kts file:

```
plugins {
    `kotlin-dsl` // Is needed to turn our build logic written in Kotlin into the Gradle Plugin
}

repositories {
    gradlePluginPortal() // To use 'maven-publish' and 'signing' plugins in our own plugin
}
```

3. In the convention-plugins directory, create a src/main/kotlin/convention.publication.gradle.kts file to store all the publication logic.

4. Add all the required logic in the new file. Be sure to make changes to match your project configuration and where explicitly noted by angle brackets (i.e. <replace-me>):

   import org.gradle.api.publish.maven.MavenPublication import org.gradle.api.tasks.bundling.Jar import org.gradle.kotlin.dsl.`maven-publish` import org.gradle.kotlin.dsl.signing import java.util.* plugins { `maven-publish` signing } // Stub secrets to let the project sync and build without the publication values set up ext["signing.keyId"] = null ext["signing.password"] = null ext["signing.secretKeyRingFile"] = null ext["ossrhUsername"] = null ext["ossrhPassword"] = null // Grabbing secrets from local.properties file or from environment variables, which could be used on CI val secretPropsFile = project.rootProject.file("local.properties") if (secretPropsFile.exists()) { secretPropsFile.reader().use { Properties().apply { load(it) } }.onEach { (name, value) -> ext[name.toString()] = value } } else { ext["signing.keyId"] = System.getenv("SIGNING_KEY_ID") ext["signing.password"] = System.getenv("SIGNING_PASSWORD") ext["signing.secretKeyRingFile"] = System.getenv("SIGNING_SECRET_KEY_RING_FILE") ext["ossrhUsername"] = System.getenv("OSSRH_USERNAME") ext["ossrhPassword"] = System.getenv("OSSRH_PASSWORD") } val javadocJar by tasks.registering(Jar::class) { archiveClassifier.set("javadoc") } fun getExtraString(name: String) = ext[name]?.toString() publishing { // Configure maven central repository repositories { maven { name = "sonatype" setUrl("https://s01.oss.sonatype.org/service/local/staging/deploy/maven2/") credentials { username = getExtraString("ossrhUsername") password = getExtraString("ossrhPassword") } } } // Configure all publications publications.withType<MavenPublication> { // Stub javadoc.jar artifact artifact(javadocJar.get()) // Provide artifacts information requited by Maven Central pom { name.set("MPP Sample library") description.set("Sample Kotlin Multiplatform library (jvm + ios + js) test") url.set("https://github.com/<your-github-repo>/mpp-sample-lib") licenses { license { name.set("MIT") url.set("https://opensource.org/licenses/MIT") } } developers { developer { id.set("<your-github-profile>") name.set("<your-name>") email.set("<your-email>") } } scm { url.set("https://github.com/<your-github-repo>/mpp-sample-lib") } } } } // Signing artifacts. Signing.* extra properties values will be used signing { sign(publishing.publications) }
   Applying just maven-publish is enough for publishing to the local Maven repository, but not to Maven Central. In the provided script, you get the credentials from local.properties or environment variables, do all the required configuration in the publishing section, and sign your publications with the signing plugin.

5. Go back to your library project. To ask Gradle to prebuild your plugins, update the root settings.gradle.kts with the following:

```
rootProject.name = "multiplatform-lib" // your project name
includeBuild("convention-plugins")
```

6. Now, you can apply this logic in the library's build.script. In the plugins section, replace maven-publish with conventional.publication:

```
plugins {
    kotlin("multiplatform") version "1.9.0"
    id("convention.publication")
}
```

7. Create a local.properties file within your library's root directory with all the necessary credentials and make sure to add it to your .gitignore:

```
# The GPG key pair ID (last 8 digits of its fingerprint)
signing.keyId=...
# The passphrase of the key pair
signing.password=...
# Private key you exported earlier
signing.secretKeyRingFile=...
# Your credentials for the Jira account
ossrhUsername=...
ossrhPassword=...
```

8. Run ./gradlew clean and sync the project.

New Gradle tasks related to the Sonatype repository should appear in the publishing group – that means that everything is ready for you to publish your library.

Publish your library to Maven Central

To upload your library to the Sonatype repository, run the following program:

```
./gradlew publishAllPublicationsToSonatypeRepository
```

The staging repository will be created, and all the artifacts for all publications will be uploaded to that repository. All it's left to do is to check that all the artifacts you wanted to upload have made it there and to press the release button.

These steps are described in the Your first release section. In short, you need to:

1. Go to https://s01.oss.sonatype.org and log in using your credentials in Sonatype Jira.
2. Find your repository in the Staging repositories section.
3. Close it.
4. Release the library.
5. To activate the sync to Maven Central, go back to the Jira issue you created and leave a comment saying that you've released your first component. This step is only needed if it's your first release.

Soon your library will be available at https://repo1.maven.org/maven2, and other developers will be able to add it as a dependency. In a couple of hours, other developers will be able to find it using Maven Central Repository Search.

Add a dependency on the published library

You can add your library to other multiplatform projects as a dependency.

In the build.gradle.kts file, add mavenLocal() or MavenCentral() (if the library was published to the external repository) and add a dependency on your library:

```
repositories {
    mavenCentral()
    mavenLocal()
}

kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.base64:multiplatform-lib:1.0.0")
            }
        }
    }
}
```

The implementation dependency consists of:

- The group ID and version — specified earlier in the build.gradle.kts file
- The artifact ID — by default, it's your project's name specified in the settings.gradle.kts file

For more details, see the Gradle documentation on the maven-publish plugin.

What's next?

- Learn more about publishing multiplatform libraries.
- Learn more about Kotlin Multiplatform.
- Create your first cross-platform mobile application – tutorial.

Publishing multiplatform libraries

You can publish a multiplatform library to a local Maven repository with the maven-publish Gradle plugin. Specify the group, version, and the repositories where the library should be published. The plugin creates publications automatically.

```
plugins {
    //...
    id("maven-publish")
}
```

```
group = "com.example"
version = "1.0"

publishing {
    repositories {
        maven {
            //...
        }
    }
}
```

To get hands-on experience, as well as learn how to publish a multiplatform library to the external Maven Central repository, see the Create and publish a multiplatform library tutorial.

> You can also publish a multiplatform library to a GitHub repository. For more information, see GitHub's documentation on GitHub packages.

Structure of publications

When used with maven-publish, the Kotlin plugin automatically creates publications for each target that can be built on the current host, except for the Android target, which needs an additional step to configure publishing.

Publications of a multiplatform library include an additional root publication kotlinMultiplatform that stands for the whole library and is automatically resolved to the appropriate platform-specific artifacts when added as a dependency to the common source set. Learn more about adding dependencies.

This kotlinMultiplatform publication includes metadata artifacts and references the other publications as its variants.

> Some repositories, such as Maven Central, require that the root module contains a JAR artifact without a classifier, for example kotlinMultiplatform-1.0.jar.
> The Kotlin Multiplatform plugin automatically produces the required artifact with the embedded metadata artifacts.
> This means you don't have to customize your build by adding an empty artifact to the root module of your library to meet the repository's requirements.

The kotlinMultiplatform publication may also need the sources and documentation artifacts if that is required by the repository. In that case, add those artifacts by using artifact(...) in the publication's scope.

Avoid duplicate publications

To avoid duplicate publications of modules that can be built on several platforms (like JVM and JS), configure the publishing tasks for these modules to run conditionally.

You can detect the platform in the script, introduce a flag such as isMainHost and set it to true for the main target platform. Alternatively, you can pass the flag from an external source, for example, from CI configuration.

This simplified example ensures that publications are only uploaded when isMainHost=true is passed. This means that a publication that can be published from multiple platforms will be published only once – from the main host.

Kotlin

```kotlin
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()
    val publicationsFromMainHost =
        listOf(jvm(), js()).map { it.name } + "kotlinMultiplatform"
    publishing {
        publications {
            matching { it.name in publicationsFromMainHost }.all {
                val targetPublication = this@all
                tasks.withType<AbstractPublishToMaven>()
                        .matching { it.publication == targetPublication }
                        .configureEach { onlyIf { findProperty("isMainHost") == "true" } }
            }
        }
```

```
    }
}
```

Groovy

```
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()
    def publicationsFromMainHost =
        [jvm(), js()].collect { it.name } + "kotlinMultiplatform"
    publishing {
        publications {
            matching { it.name in publicationsFromMainHost }.all { targetPublication ->
                tasks.withType(AbstractPublishToMaven)
                        .matching { it.publication == targetPublication }
                        .configureEach { onlyIf { findProperty("isMainHost") == "true" } }
            }
        }
    }
}
```

By default, each publication includes a sources JAR that contains the sources used by the main compilation of the target.

Publish an Android library

To publish an Android library, you need to provide additional configuration.

By default, no artifacts of an Android library are published. To publish artifacts produced by a set of Android variants, specify the variant names in the Android target block:

```
kotlin {
    android {
        publishLibraryVariants("release", "debug")
    }
}
```

The example works for Android libraries without product flavors. For a library with product flavors, the variant names also contain the flavors, like fooBarDebug or fooBazRelease.

The default publishing setup is as follows:

- If the published variants have the same build type (for example, all of them are release ordebug), they will be compatible with any consumer build type.
- If the published variants have different build types, then only the release variants will be compatible with consumer build types that are not among the published variants. All other variants (such as debug) will only match the same build type on the consumer side, unless the consumer project specifies the matching fallbacks.

If you want to make every published Android variant compatible with only the same build type used by the library consumer, set this Gradle property: kotlin.android.buildTypeAttribute.keep=true.

You can also publish variants grouped by the product flavor, so that the outputs of the different build types are placed in a single module, with the build type becoming a classifier for the artifacts (the release build type is still published with no classifier). This mode is disabled by default and can be enabled as follows:

```
kotlin {
    android {
        publishLibraryVariantsGroupedByFlavor = true
    }
}
```

> It is not recommended that you publish variants grouped by the product flavor in case they have different dependencies, as those will be merged into one dependencies list.

Share code on platforms

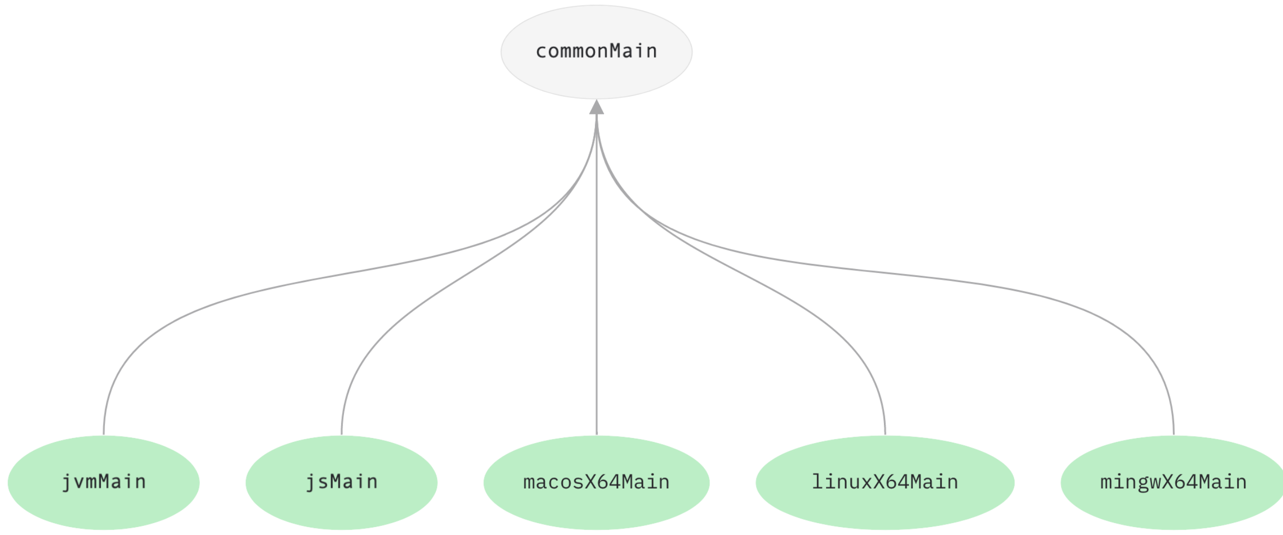
With Kotlin Multiplatform, you can share the code using the mechanisms Kotlin provides:

- Share code among all platforms used in your project. Use it for sharing the common business logic that applies to all platforms.
- Share code among some platforms included in your project but not all. You can reuse code in similar platforms with a help of the hierarchical structure.

If you need to access platform-specific APIs from the shared code, use the Kotlin mechanism of expected and actual declarations.

Share code on all platforms

If you have business logic that is common for all platforms, you don't need to write the same code for each platform – just share it in the common source set.



Code shared for all platforms

Some dependencies for source sets are set by default. You don't need to specify any dependsOn relations manually:

- For all platform-specific source sets that depend on the common source set, such as jvmMain, macosX64Main, and others.
- Between the main and test source sets of a particular target, such as androidMain and androidTest.

If you need to access platform-specific APIs from the shared code, use the Kotlin mechanism of expected and actual declarations.

Share code on similar platforms

You often need to create several native targets that could potentially reuse a lot of the common logic and third-party APIs.

For example, in a typical multiplatform project targeting iOS, there are two iOS-related targets: one is for iOS ARM64 devices, the other is for the x64 simulator. They have separate platform-specific source sets, but in practice there is rarely a need for different code for the device and simulator, and their dependencies are much the same. So iOS-specific code could be shared between them.

Evidently, in this setup it would be desirable to have a shared source set for two iOS targets, with Kotlin/Native code that could still directly call any of the APIs that are common to both the iOS device and the simulator.

In this case, you can share code across native targets in your project using the hierarchical structure using one of the following ways:

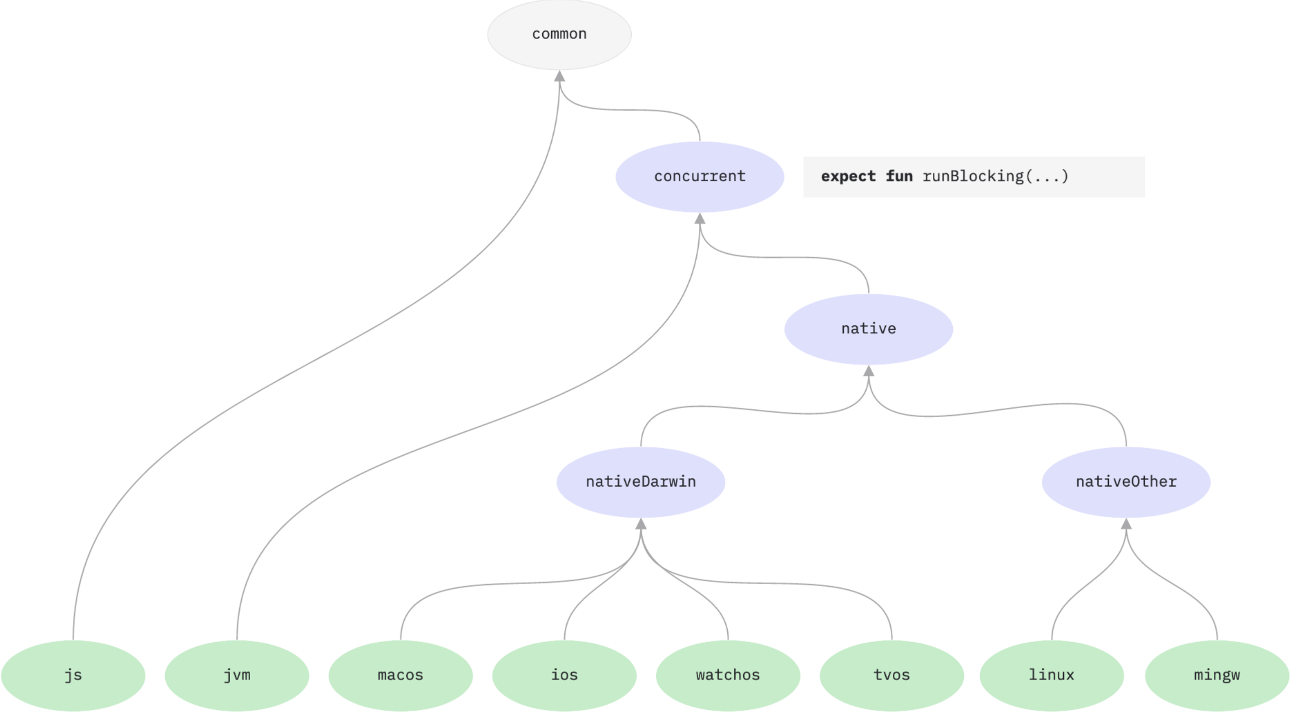
- Enabling default target hierarchy
- Using target shortcuts
- Configuring the hierarchical structure manually

Learn more about sharing code in libraries and connecting platform-specific libraries.

Share code in libraries

Thanks to the hierarchical project structure, libraries can also provide common APIs for a subset of targets. When a library is published, the API of its intermediate source sets is embedded into the library artifacts along with information about the project structure. When you use this library, the intermediate source sets of your project access only those APIs of the library which are available to the targets of each source set.

For example, check out the following source set hierarchy from the kotlinx.coroutines repository:



Library hierarchical structure

The concurrent source set declares the function runBlocking and is compiled for the JVM and the native targets. Once the kotlinx.coroutines library is updated and published with the hierarchical project structure, you can depend on it and call runBlocking from a source set that is shared between the JVM and native targets since it matches the "targets signature" of the library's concurrent source set.

Connect platform-specific libraries

Platform-specific libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) are available in shared source sets by default. This helps you share more native code without being limited by platform-specific dependencies.

In addition, you can enable the support for third-party libraries consumed with the cinterop mechanism. To do that, add the following property to your gradle.properties:

```
kotlin.mpp.enableCInteropCommonization=true
```

What's next?

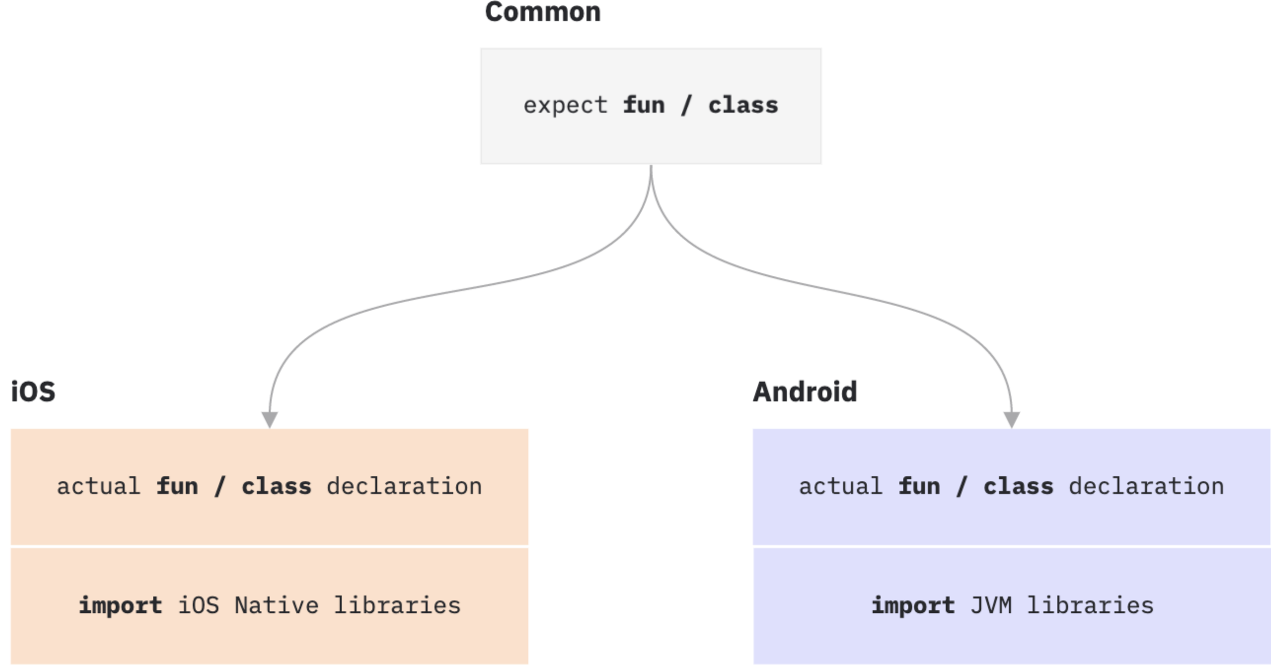
- Check out examples of code sharing using the Kotlin mechanism of expect and actual declarations
- Learn more about hierarchical project structure
- See our recommendations on naming source files in multiplatform projects

Connect to platform-specific APIs

The expect/actual feature is in Beta. It is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you will have to make.

If you're developing a multiplatform application that needs to access platform-specific APIs that implement the required functionality (for example, generating a UUID), use the Kotlin mechanism of expected and actual declarations.

With this mechanism, a common source set defines an expected declaration, and platform source sets must provide the actual declaration that corresponds to the expected declaration. This works for most Kotlin declarations, such as functions, classes, interfaces, enumerations, properties, and annotations.



Expect/actual declarations in common and platform-specific modules

The compiler ensures that every declaration marked with the expect keyword in the common module has the corresponding declarations marked with the actual keyword in all platform modules. The IDE provides tools that help you create the missing actual declarations.

Use expected and actual declarations only for Kotlin declarations that have platform-specific dependencies. Implementing as much functionality as possible in the shared module is better, even if doing so takes more time.

Don't overuse expected and actual declarations – in some cases, an interface may be a better choice because it is more flexible and easier to test.

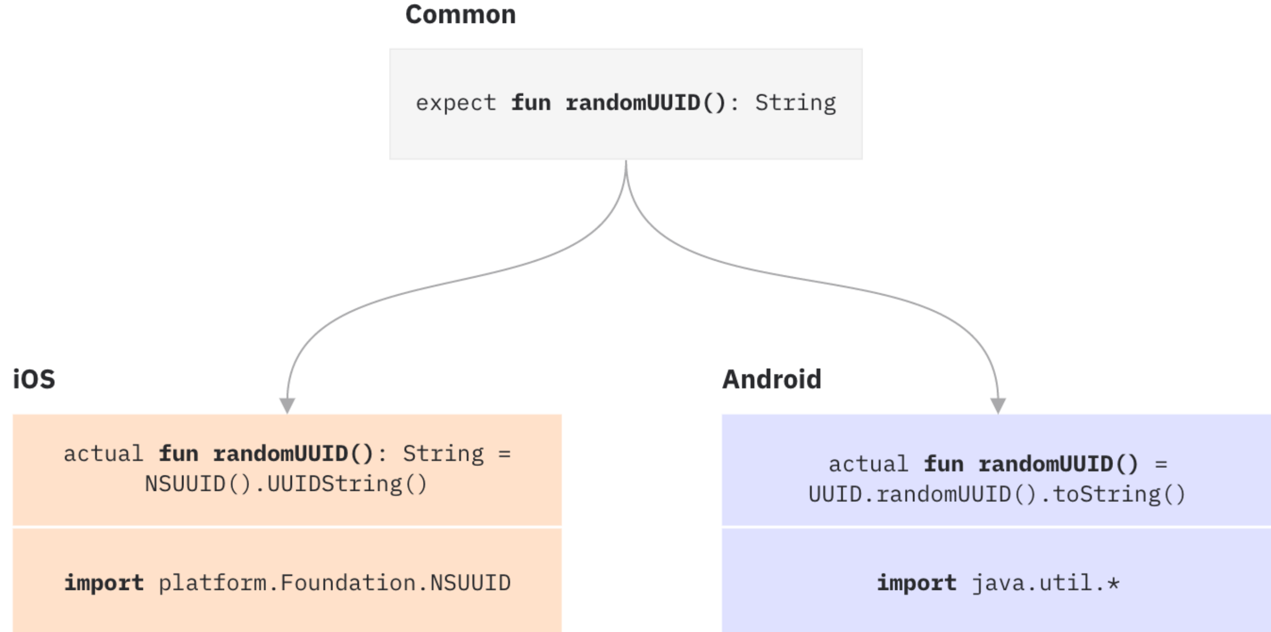
Learn how to add dependencies on platform-specific libraries.

Examples

For simplicity, the following examples use intuitive target names, like iOS and Android. However, in your Gradle build files, you need to use a specific target name from the list of supported targets.

Generate a UUID

Let's assume that you are developing iOS and Android applications using Kotlin Multiplatform and you want to generate a universally unique identifier (UUID):



Expect/actual declarations for getting the UUID

For this purpose, declare the expected function randomUUID() with the expect keyword in the common module. Don't include any implementation code.

```
// Common
expect fun randomUUID(): String
```

In each platform-specific module (iOS and Android), provide the actual implementation for the function randomUUID() expected in the common module. Use the actual keyword to mark the actual implementation.

The following examples show the implementation of this for Android and iOS. Platform-specific code uses the actual keyword and the expected name for the function.

```
// Android
import java.util.*

actual fun randomUUID() = UUID.randomUUID().toString()
```

```
// iOS
import platform.Foundation.NSUUID

actual fun randomUUID(): String = NSUUID().UUIDString()
```

Implement a logging framework

Another example of code sharing and interaction between the common and platform logic, JS and JVM in this case, in a minimalistic logging framework:

```
// Common
enum class LogLevel {
    DEBUG, WARN, ERROR
}

internal expect fun writeLogMessage(message: String, logLevel: LogLevel)

fun logDebug(message: String) = writeLogMessage(message, LogLevel.DEBUG)
fun logWarn(message: String) = writeLogMessage(message, LogLevel.WARN)
fun logError(message: String) = writeLogMessage(message, LogLevel.ERROR)
```

```
// JVM
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    println("[$logLevel]: $message")
```

```
}
```

For JavaScript, a completely different set of APIs is available, and the actual declaration will look like this.

```
// JS
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    when (logLevel) {
        LogLevel.DEBUG -> console.log(message)
        LogLevel.WARN -> console.warn(message)
        LogLevel.ERROR -> console.error(message)
    }
}
```

Send and receive messages from a WebSocket

Consider developing a chat platform for iOS and Android using Kotlin Multiplatform. Let's see how you can implement sending and receiving messages from a WebSocket.

For this purpose, define a common logic that you don't need to duplicate in all platform modules – just add it once to the common module. However, the actual implementation of the WebSocket class differs from platform to platform. That's why you should use expect/actual declarations for this class.

In the common module, declare the expected class PlatformSocket() with the expect keyword. Don't include any implementation code.

```
//Common
internal expect class PlatformSocket(
    url: String
) {
    fun openSocket(listener: PlatformSocketListener)
    fun closeSocket(code: Int, reason: String)
    fun sendMessage(msg: String)
}
interface PlatformSocketListener {
    fun onOpen()
    fun onFailure(t: Throwable)
    fun onMessage(msg: String)
    fun onClosing(code: Int, reason: String)
    fun onClosed(code: Int, reason: String)
}
```

In each platform-specific module (iOS and Android), provide the actual implementation for the class PlatformSocket() expected in the common module. Use the actual keyword to mark the actual implementation.

The following examples show the implementation of this for Android and iOS.

```
//Android
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.Response
import okhttp3.WebSocket

internal actual class PlatformSocket actual constructor(url: String) {
    private val socketEndpoint = url
    private var webSocket: WebSocket? = null
    actual fun openSocket(listener: PlatformSocketListener) {
        val socketRequest = Request.Builder().url(socketEndpoint).build()
        val webClient = OkHttpClient().newBuilder().build()
        webSocket = webClient.newWebSocket(
            socketRequest,
            object : okhttp3.WebSocketListener() {
                override fun onOpen(webSocket: WebSocket, response: Response) = listener.onOpen()
                override fun onFailure(webSocket: WebSocket, t: Throwable, response: Response?) = listener.onFailure(t)
                override fun onMessage(webSocket: WebSocket, text: String) = listener.onMessage(text)
                override fun onClosing(webSocket: WebSocket, code: Int, reason: String) = listener.onClosing(code, reason)
                override fun onClosed(webSocket: WebSocket, code: Int, reason: String) = listener.onClosed(code, reason)
            }
        )
    }
    actual fun closeSocket(code: Int, reason: String) {
        webSocket?.close(code, reason)
        webSocket = null
    }
    actual fun sendMessage(msg: String) {
        webSocket?.send(msg)
    }
```

```
}
```

Android implementation uses the third-party library OkHttp. Add the corresponding dependency to build.gradle(.kts) in the shared module:

Kotlin

```kotlin
sourceSets {
    val androidMain by getting {
        dependencies {
            implementation("com.squareup.okhttp3:okhttp:$okhttp_version")
        }
    }
}
```

Groovy

```groovy
commonMain {
    dependencies {
        implementation "com.squareup.okhttp3:okhttp:$okhttp_version"
    }
}
```

iOS implementation uses NSURLSession from the standard Apple SDK and doesn't require additional dependencies.

```kotlin
//iOS
import platform.Foundation.*
import platform.darwin.NSObject

internal actual class PlatformSocket actual constructor(url: String) {
    private val socketEndpoint = NSURL.URLWithString(url)!!
    private var webSocket: NSURLSessionWebSocketTask? = null
    actual fun openSocket(listener: PlatformSocketListener) {
        val urlSession = NSURLSession.sessionWithConfiguration(
            configuration = NSURLSessionConfiguration.defaultSessionConfiguration(),
            delegate = object : NSObject(), NSURLSessionWebSocketDelegateProtocol {
                override fun URLSession(
                    session: NSURLSession,
                    webSocketTask: NSURLSessionWebSocketTask,
                    didOpenWithProtocol: String?
                ) {
                    listener.onOpen()
                }
                override fun URLSession(
                    session: NSURLSession,
                    webSocketTask: NSURLSessionWebSocketTask,
                    didCloseWithCode: NSURLSessionWebSocketCloseCode,
                    reason: NSData?
                ) {
                    listener.onClosed(didCloseWithCode.toInt(), reason.toString())
                }
            },
            delegateQueue = NSOperationQueue.currentQueue()
        )
        webSocket = urlSession.webSocketTaskWithURL(socketEndpoint)
        listenMessages(listener)
        webSocket?.resume()
    }
    private fun listenMessages(listener: PlatformSocketListener) {
        webSocket?.receiveMessageWithCompletionHandler { message, nsError ->
            when {
                nsError != null -> {
                    listener.onFailure(Throwable(nsError.description))
                }
                message != null -> {
                    message.string?.let { listener.onMessage(it) }
                }
            }
            listenMessages(listener)
        }
    }
    actual fun closeSocket(code: Int, reason: String) {
        webSocket?.cancelWithCloseCode(code.toLong(), null)
        webSocket = null
    }
```

```
    actual fun sendMessage(msg: String) {
        val message = NSURLSessionWebSocketMessage(msg)
        webSocket?.sendMessage(message) { err ->
            err?.let { println("send $msg error: $it") }
        }
    }
}
```

And here is the common logic in the common module that uses the platform-specific class PlatformSocket().

```
//Common
class AppSocket(url: String) {
    private val ws = PlatformSocket(url)
    var socketError: Throwable? = null
        private set
    var currentState: State = State.CLOSED
        private set(value) {
            field = value
            stateListener?.invoke(value)
        }
    var stateListener: ((State) -> Unit)? = null
        set(value) {
            field = value
            value?.invoke(currentState)
        }
    var messageListener: ((msg: String) -> Unit)? = null
    fun connect() {
        if (currentState != State.CLOSED) {
            throw IllegalStateException("The socket is available.")
        }
        socketError = null
        currentState = State.CONNECTING
        ws.openSocket(socketListener)
    }
    fun disconnect() {
        if (currentState != State.CLOSED) {
            currentState = State.CLOSING
            ws.closeSocket(1000, "The user has closed the connection.")
        }
    }
    fun send(msg: String) {
        if (currentState != State.CONNECTED) throw IllegalStateException("The connection is lost.")
        ws.sendMessage(msg)
    }
    private val socketListener = object : PlatformSocketListener {
        override fun onOpen() {
            currentState = State.CONNECTED
        }
        override fun onFailure(t: Throwable) {
            socketError = t
            currentState = State.CLOSED
        }
        override fun onMessage(msg: String) {
            messageListener?.invoke(msg)
        }
        override fun onClosing(code: Int, reason: String) {
            currentState = State.CLOSING
        }
        override fun onClosed(code: Int, reason: String) {
            currentState = State.CLOSED
        }
    }
    enum class State {
        CONNECTING,
        CONNECTED,
        CLOSING,
        CLOSED
    }
}
```

Rules for expected and actual declarations

The main rules regarding expected and actual declarations are:

- An expected declaration is marked with the expect keyword; the actual declaration is marked with the actual keyword.
- expect and actual declarations have the same name and are located in the same package (have the same fully qualified name).

- expect declarations never contain any implementation code and are abstract by default.
- In interfaces, functions in expect declarations cannot have bodies, but their actual counterparts can be non-abstract and have a body. It allows the inheritors not to implement a particular function.

  To indicate that common inheritors don't need to implement a function, mark it as open. All its actual implementations will be required to have a body:

```
// Common
expect interface Mascot {
    open fun display(): String
}

class MascotImpl : Mascot {
    // it's ok not to implement `display()`: all `actual`s are guaranteed to have a default implementation
}

// Platform-specific
actual interface Mascot {
    actual fun display(): String {
        TODO()
    }
}
```

During each platform compilation, the compiler ensures that every declaration marked with the expect keyword in the common or intermediate source set has the corresponding declarations marked with the actual keyword in all platform source sets. The IDE provides tools that help you create the missing actual declarations.

If you have a platform-specific library that you want to use in shared code while providing your own implementation for another platform, you can provide a typealias to an existing class as the actual declaration:

```
expect class AtomicRef<V>(value: V) {
    fun get(): V
    fun set(value: V)
    fun getAndSet(value: V): V
    fun compareAndSet(expect: V, update: V): Boolean
}
```

```
actual typealias AtomicRef<V> = java.util.concurrent.atomic.AtomicReference<V>
```

Hierarchical project structure

Multiplatform projects support hierarchical structures. This means you can arrange a hierarchy of intermediate source sets for sharing the common code among some, but not all, supported targets. Using intermediate source sets has some important advantages:

- If you're a library author and you want to provide a specialized API, you can use an intermediate source set for some, but not all, targets – for example, an intermediate source set for Kotlin/Native targets but not for Kotlin/JVM ones.
- If you want to use platform-dependent libraries in your project, you can use an intermediate source set to use that specific API in several native targets. For example, you can have access to iOS-specific dependencies, such as Foundation, when sharing code across all iOS targets.
- Some libraries aren't available for particular platforms. Specifically, native libraries are only available for source sets that compile to Kotlin/Native. Using an intermediate source set will solve this issue.

The Kotlin toolchain ensures that each source set has access only to the API that is available for all targets to which that source set compiles. This prevents cases like using a Windows-specific API and then compiling it to macOS, resulting in linkage errors or undefined behavior at runtime.

There are 3 ways to create a target hierarchy:

- Specify all targets and enable the default hierarchy
- Use target shortcuts available for typical cases
- Manually declare and connect the source sets

Default hierarchy

> The default target hierarchy is Experimental. It may be changed in future Kotlin releases without prior notice. For Kotlin Gradle build scripts, opting in is required with @OptIn(ExperimentalKotlinGradlePluginApi::class).

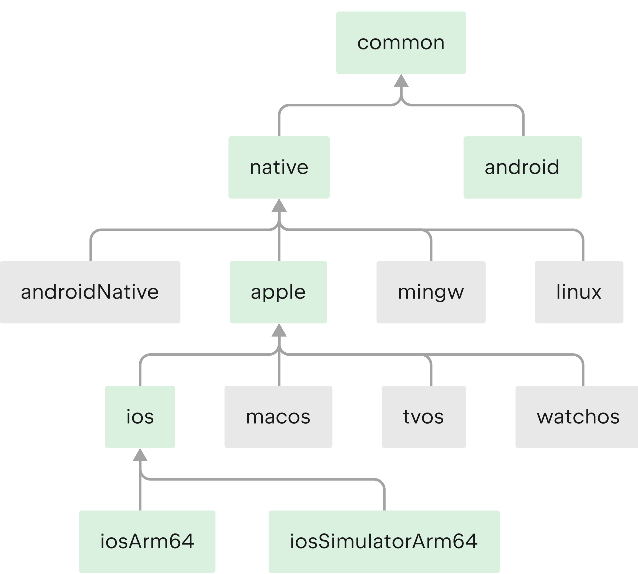
Starting with Kotlin 1.8.20, you can set up a source set hierarchy in your multiplatform projects with the default target hierarchy. It's a template for all possible targets and their shared source sets hardcoded in the Kotlin Gradle plugin.

Set up your project

To set up a hierarchy, call targetHierarchy.default() in the kotlin block of your build.gradle(.kts) file and list all of the targets you need. For example:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
kotlin {
    // Enable the default target hierarchy:
    targetHierarchy.default()

    android()
    iosArm64()
    iosSimulatorArm64()
}
```

When you declare the final targets android, iosArm64, and iosSimulatorArm64 in your code, the Kotlin Gradle plugin finds suitable shared source sets from the template and creates them for you. The resulting hierarchy looks like this:



An example of using the default target hierarchy

Green source sets are actually created and present in the project, while gray ones from the default template are ignored. The Kotlin Gradle plugin hasn't created the watchos source set, for example, because there are no watchOS targets in the project.

If you add a watchOS target, like watchosArm64, the watchos source set is created, and the code from the apple, native, and common source sets is compiled to watchosArm64 as well.

> In this example, the apple and native source sets compile only to the iosArm64 and iosSimulatorArm64 targets. Despite their names, they have access to the full iOS API. This can be counter-intuitive for source sets like native, as you might expect that only APIs available on all native targets are accessible in this source set. This behavior may change in the future.

Adjust the resulting hierarchy

You can further configure the resulting hierarchy manually using the dependsOn relation. To do so, apply the by getting construction for the source sets created with targetHierarchy.default().

Consider this example of a project with a source set shared between the jvm and native targets only:

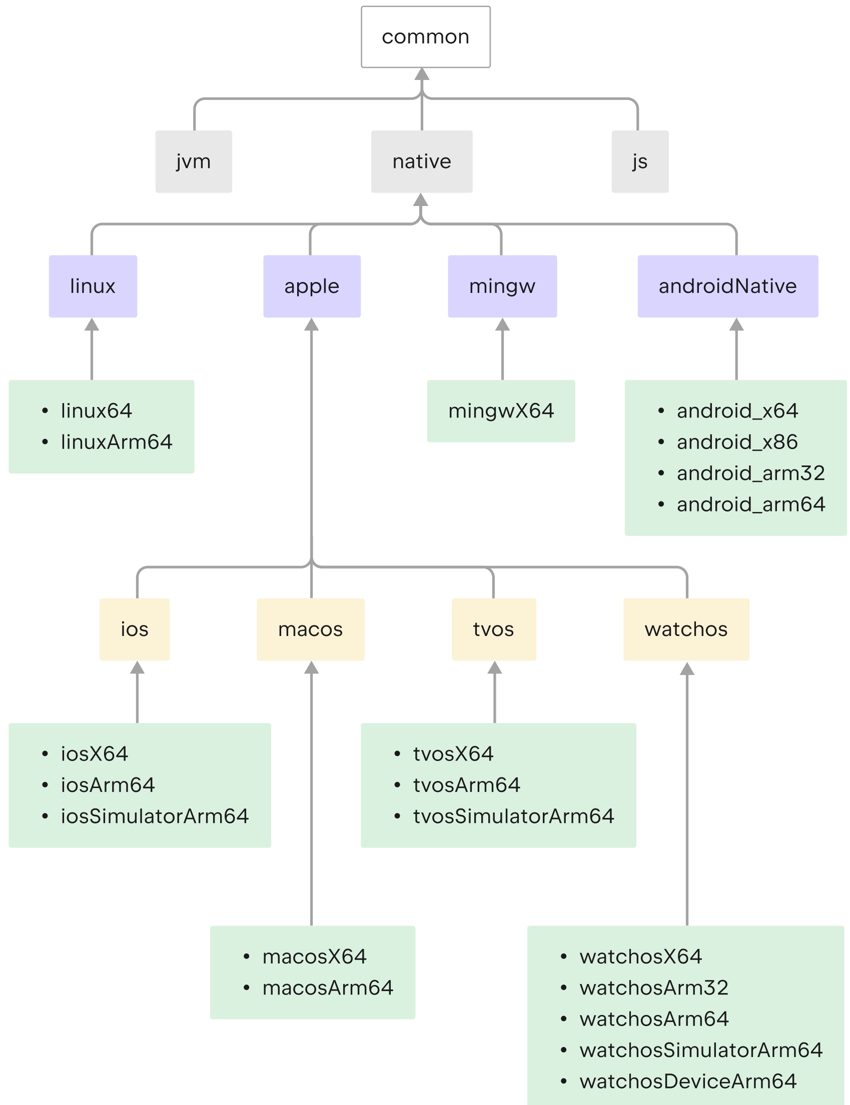
```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
kotlin {
    // Enable the default target hierarchy:
    targetHierarchy.default()

    jvm()
    iosArm64()
    // the rest of the necessary targets...

    sourceSets {
        val commonMain by getting

        val jvmAndNativeMain by creating {
            dependsOn(commonMain)
        }

        val nativeMain by getting {
            dependsOn(jvmAndNativeMain)
        }

        val jvmMain by getting {
            dependsOn(jvmAndNativeMain)
        }
    }
}
```

It can be cumbersome to remove dependsOn relations that are automatically created by the targetHierarchy.default() call. In that case, use an entirely manual configuration instead of calling the default hierarchy.

> We're currently working on an API to create your own target hierarchies. It will be useful for projects whose hierarchy configurations are significantly different from the default template.
>
> This API is not ready yet, but if you're eager to try it, look into the targetHierarchy.custom { ... } block and the declaration of targetHierarchy.default() as an example. Keep in mind that this API is still in development. It might not be tested, and can change in further releases.

See the full hierarchy template

When you declare the targets to which your project compiles, the plugin picks the shared source sets based on the specified targets from the template and creates them in your project.



Default target hierarchy

> This example only shows the production part of the project, omitting the Main suffix (for example, using common instead of commonMain). However, everything is the same for *Test sources as well.

Target shortcuts

The Kotlin Multiplatform plugin provides some predefined target shortcuts for creating structures for common target combinations:

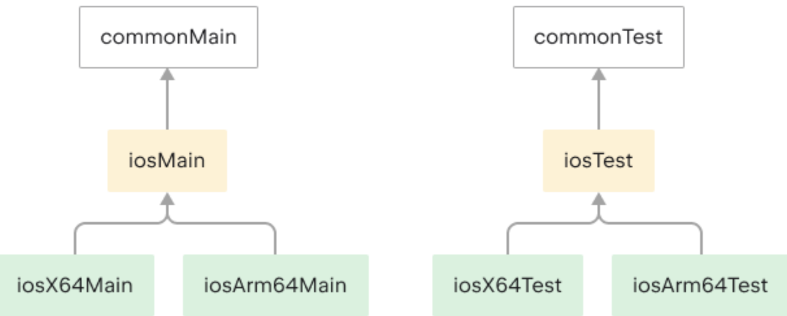
Target shortcut	Targets
ios	iosArm64, iosX64
watchos	watchosArm32, watchosArm64, watchosX64

Target shortcut	Targets
tvos	tvosArm64, tvosX64

All shortcuts create similar hierarchical structures in the code. For example, you can use theios() shortcut to create a multiplatform project with 2 iOS-related targets, iosArm64 and iosX64, and a shared source set:

```
kotlin {
    ios() // iOS device and simulator targets; iosMain and iosTest source sets
}
```

In this case, the hierarchical structure includes the intermediate source sets iosMain and iosTest, which are used by the platform-specific source sets:



Code shared for iOS targets

The resulting hierarchical structure will be equivalent to the code below:

Kotlin

```
kotlin {
    iosX64()
    iosArm64()

    sourceSets {
        val commonMain by getting
        val iosX64Main by getting
        val iosArm64Main by getting
        val iosMain by creating {
            dependsOn(commonMain)
            iosX64Main.dependsOn(this)
            iosArm64Main.dependsOn(this)
        }
    }
}
```

Groovy

```
kotlin {
    iosX64()
    iosArm64()

    sourceSets {
        iosMain {
            dependsOn(commonMain)
            iosX64Main.dependsOn(it)
            iosArm64Main.dependsOn(it)
        }
    }
}
```

Target shortcuts and ARM64 (Apple Silicon) simulators

The ios, watchos, and tvos target shortcuts don't include the simulator targets for ARM64 (Apple Silicon) platforms: iosSimulatorArm64, watchosSimulatorArm64, and tvosSimulatorArm64. If you use the target shortcuts and want to build the project for an Apple Silicon simulator, make the following adjustment to the build script:

1. Add the *SimulatorArm64 simulator target you need.

2. Connect the simulator target with the shortcut using the dependsOn relation between source sets.

Kotlin

```
kotlin {
    ios()
    // Add the ARM64 simulator target
    iosSimulatorArm64()

    val iosMain by sourceSets.getting
    val iosTest by sourceSets.getting
    val iosSimulatorArm64Main by sourceSets.getting
    val iosSimulatorArm64Test by sourceSets.getting

    // Set up dependencies between the source sets
    iosSimulatorArm64Main.dependsOn(iosMain)
    iosSimulatorArm64Test.dependsOn(iosTest)
}
```

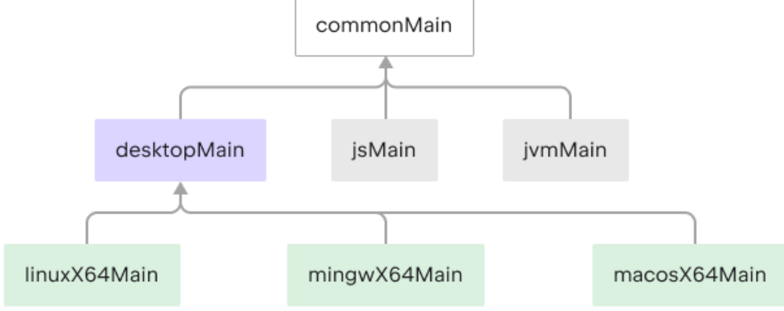
Groovy

```
kotlin {
    ios()
    // Add the ARM64 simulator target
    iosSimulatorArm64()

    // Set up dependencies between the source sets
    sourceSets {
        // ...
        iosSimulatorArm64Main {
            dependsOn(iosMain)
        }
        iosSimulatorArm64Test {
            dependsOn(iosTest)
        }
    }
}
```

Manual configuration

You can manually introduce an intermediate source in the source set structure. It will hold the shared code for several targets.

For example, here’s what to do if you want to share code among native Linux, Windows, and macOS targets (linuxX64, mingwX64, and macosX64):



Manually configured hierarchical structure

1. Add the intermediate source set desktopMain, which holds the shared logic for these targets.
2. Specify the source set hierarchy using the dependsOn relation.

The resulting hierarchical structure will look like this:

Kotlin

```kotlin
kotlin {
    linuxX64()
    mingwX64()
    macosX64()

    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain.get())
        }
        val linuxX64Main by getting {
            dependsOn(desktopMain)
        }
        val mingwX64Main by getting {
            dependsOn(desktopMain)
        }
        val macosX64Main by getting {
            dependsOn(desktopMain)
        }
    }
}
```

Groovy

```groovy
kotlin {
    linuxX64()
    mingwX64()
    macosX64()

    sourceSets {
        desktopMain {
            dependsOn(commonMain.get())
        }
        linuxX64Main {
            dependsOn(desktopMain)
        }
        mingwX64Main {
            dependsOn(desktopMain)
        }
        macosX64Main {
            dependsOn(desktopMain)
        }
    }
}
```

You can have a shared source set for the following combinations of targets:

- JVM or Android + JS + Native
- JVM or Android + Native
- JS + Native
- JVM or Android + JS
- Native

Kotlin doesn't currently support sharing a source set for these combinations:

- Several JVM targets
- JVM + Android targets
- Several JS targets

If you need to access platform-specific APIs from a shared native source set, IntelliJ IDEA will help you detect common declarations that you can use in the shared

native code. For other cases, use the Kotlin mechanism of expected and actual declarations.

Android source set layout

The new Android source set layout was introduced in Kotlin 1.8.0 and became default in 1.9.0. Follow this guide to understand the key differences between the deprecated and the new layout and migrate your projects.

> You don't need to implement all the suggestions. Only those that are applicable to your particular projects.

Check the compatibility

The new layout requires Android Gradle plugin 7.0 or later and is supported in Android Studio 2022.3 and later. Check your version of the Android Gradle plugin and upgrade if necessary.

Rename Kotlin source sets

If applicable, rename source sets in your project, following this pattern:

Previous source set layout	New source set layout
targetName + AndroidSourceSet.name	targetName + AndroidVariantType

{AndroidSourceSet.name} maps to {KotlinSourceSet.name} as follows:

	Previous source set layout	New source set layout
main	androidMain	androidMain
test	androidTest	androidUnitTest
androidTest	androidAndroidTest	androidInstrumentedTest

Move source files

If applicable, move your source files to the new directories, following this pattern:

Previous source set layout	New source set layout
The layout had additional /kotlin SourceDirectories	src/{KotlinSourceSet.name}/kotlin

{AndroidSourceSet.name} maps to {SourceDirectories included} as follows:

	Previous source set layout	New source set layout

	Previous source set layout	New source set layout
main	src/androidMain/kotlin src/main/kotlin src/main/java	src/androidMain/kotlin src/main/kotlin src/main/java
test	src/androidTest/kotlin src/test/kotlin src/test/java	src/androidUnitTest/kotlin src/test/kotlin src/test/java
androidTest	src/androidAndroidTest/kotlin src/androidTest/java	src/androidInstrumentedTest/kotlin src/androidTest/java, src/androidTest/kotlin

Move the AndroidManifest.xml file

If you have the AndroidManifest.xml file in your project, move it to the new directory, following this pattern:

Previous source set layout	New source set layout
src/{AndroidSourceSet.name}/AndroidManifest.xml	src/{KotlinSourceSet.name}/AndroidManifest.xml

{AndroidSourceSet.name} maps to {AndroidManifest.xml location} as follows:

	Previous source set layout	New source set layout
main	src/main/AndroidManifest.xml	src/androidMain/AndroidManifest.xml
debug	src/debug/AndroidManifest.xml	src/androidDebug/AndroidManifest.xml

Check the relation between Android and common tests

The new Android source set layout changes the relation between Android-instrumented tests (renamed to androidInstrumentedTest in the new layout) and common tests.

Previously, there was a default dependsOn relation between androidAndroidTest and commonTest. It meant the following:

- The code in commonTest was available in androidAndroidTest.
- expect declarations in commonTest had to have corresponding actual implementations in androidAndroidTest.
- Tests declared in commonTest were also running as Android instrumented tests.

In the new Android source set layout, the dependsOn relation is not added by default. If you prefer the previous behavior, manually declare this relation in your build.gradle.kts file:

```kotlin
kotlin {
// ...
    sourceSets {
        val commonTest by getting
        val androidInstrumentedTest by getting {
            dependsOn(commonTest)
```

```
        }
    }
}
```

Adjust the implementation of Android flavors

Previously, the Kotlin Gradle plugin eagerly created source sets that correspond to Android source sets with debug and release build types or custom flavors like demo and full. It made them accessible by constructions like val androidDebug by getting { ... }.

The new Android source set layout makes use of Android onVariants to create source sets. It makes such expressions invalid, leading to errors like org.gradle.api.UnknownDomainObjectException: KotlinSourceSet with name 'androidDebug' not found.

To work around that, use the new invokeWhenCreated() API in your build.gradle.kts file:

```
kotlin {
// ...
    @OptIn(ExperimentalKotlinGradlePluginApi::class)
    sourceSets.invokeWhenCreated("androidFreeDebug") {
        // ...
    }
}
```

Adding dependencies on multiplatform libraries

Every program requires a set of libraries to operate successfully. A Kotlin Multiplatform project can depend on multiplatform libraries that work for all target platforms, platform-specific libraries, and other multiplatform projects.

To add a dependency on a library, update your build.gradle(.kts) file in the shared directory of your project. Set a dependency of the required type (for example, implementation) in the dependencies block:

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("com.example:my-library:1.0") // library shared for all source sets
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'com.example:my-library:1.0'
            }
        }
    }
}
```

Alternatively, you can set dependencies at the top level.

Dependency on a Kotlin library

Standard library

A dependency on a standard library (stdlib) in each source set is added automatically. The version of the standard library is the same as the version of the kotlin-

multiplatform plugin.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin will select the appropriate JVM standard library depending on the compilerOptions.jvmTarget compiler option of your Gradle build script.

Learn how to change the default behavior.

Test libraries

The kotlin.test API is available for multiplatform tests. When you create a multiplatform project, the Project Wizard automatically adds test dependencies to common and platform-specific source sets.

If you didn't use the Project Wizard to create your project, you can add the dependencies manually.

kotlinx libraries

If you use a multiplatform library and need to depend on the shared code, set the dependency only once in the shared source set. Use the library base artifact name, such as kotlinx-coroutines-core.

Kotlin

```kotlin
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1'
            }
        }
    }
}
```

If you use a kotlinx library and need a platform-specific dependency, you can use platform-specific variants of libraries with suffixes such as -jvm or -js, for example, kotlinx-coroutines-core-jvm.

Kotlin

```kotlin
kotlin {
    sourceSets {
        val jvmMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.7.1")
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        jvmMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.7.1'
            }
```

```
        }
    }
}
```

Dependency on Kotlin Multiplatform libraries

You can add dependencies on libraries that have adopted Kotlin Multiplatform technology, such as SQLDelight. The authors of these libraries usually provide guides for adding their dependencies to your project.

Check out this community-maintained list of Kotlin Multiplatform libraries.

Library shared for all source sets

If you want to use a library from all source sets, you can add it only to the common source set. The Kotlin Multiplatform Mobile plugin will automatically add the corresponding parts to any other source sets.

> You cannot set dependencies on platform-specific libraries in the common source set.

Kotlin

```kotlin
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("io.ktor:ktor-client-core:2.3.2")
            }
        }
        val androidMain by getting {
            dependencies {
                // dependency to a platform part of ktor-client will be added automatically
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'io.ktor:ktor-client-core:2.3.2'
            }
        }
        androidMain {
            dependencies {
                // dependency to platform part of ktor-client will be added automatically
            }
        }
    }
}
```

Library used in specific source sets

If you want to use a multiplatform library just for specific source sets, you can add it exclusively to them. The specified library declarations will then be available only in those source sets.

> Don't use a platform-specific name in such cases, like SQLDelight native-driver in the example below. Find the exact name in the library's documentation.

Kotlin

```kotlin
kotlin {
    sourceSets {
```

```
        val commonMain by getting {
            dependencies {
                // kotlinx.coroutines will be available in all source sets
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
            }
        }
        val androidMain by getting {
            dependencies {}
        }
        val iosMain by getting {
            dependencies {
                // SQLDelight will be available only in the iOS source set, but not in Android or common
                implementation("com.squareup.sqldelight:native-driver:1.5.5")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                // kotlinx.coroutines will be available in all source sets
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1'
            }
        }
        androidMain {
            dependencies {}
        }
        iosMain {
            dependencies {
                // SQLDelight will be available only in the iOS source set, but not in Android or common
                implementation 'com.squareup.sqldelight:native-driver:1.5.5'
            }
        }
    }
}
```

Dependency on another multiplatform project

You can connect one multiplatform project to another as a dependency. To do this, simply add a project dependency to the source set that needs it. If you want to use a dependency in all source sets, add it to the common one. In this case, other source sets will get their versions automatically.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation(project(":some-other-multiplatform-module"))
            }
        }
        val androidMain by getting {
            dependencies {
                // platform part of :some-other-multiplatform-module will be added automatically
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation project(':some-other-multiplatform-module')
            }
        }
        androidMain {
```

```
            dependencies {
                // platform part of :some-other-multiplatform-module will be added automatically
            }
        }
    }
}
```

What's next?

Check out other resources on adding dependencies in multiplatform projects and learn more about:

- Adding Android dependencies
- Adding iOS dependencies

Adding Android dependencies

The workflow for adding Android-specific dependencies to a Kotlin Multiplatform module is the same as it is for pure Android projects: declare the dependency in your Gradle file and import the project. After that, you can use this dependency in your Kotlin code.

We recommend declaring Android dependencies in Multiplatform Mobile projects by adding them to a specific Android source set. For that, update your build.gradle(.kts) file in the shared directory of your project:

Kotlin

```kotlin
sourceSets["androidMain"].dependencies {
    implementation("com.example.android:app-magic:12.3")
}
```

Groovy

```groovy
sourceSets {
    androidMain {
        dependencies {
            implementation 'com.example.android:app-magic:12.3'
        }
    }
}
```

Moving what was a top-level dependency in an Android project to a specific source set in a Multiplatform Mobile project might be difficult if the top-level dependency had a non-trivial configuration name. For example, to move a debugImplementation dependency from the top level of an Android project, you'll need to add an implementation dependency to the source set named androidDebug. To minimize the effort you have to put in to deal with migration problems like this, you can add a dependencies block inside the android block:

Kotlin

```kotlin
android {
    //...
    dependencies {
        implementation("com.example.android:app-magic:12.3")
    }
}
```

Groovy

```groovy
android {
    //...
    dependencies {
        implementation 'com.example.android:app-magic:12.3'
    }
}
```

Dependencies declared here will be treated exactly the same as dependencies from the top-level block, but declaring them this way will also separate Android dependencies visually in your build script and make it less confusing.

Putting dependencies into a standalone dependencies block at the end of the script, in a way that is idiomatic to Android projects, is also supported. However, we strongly recommend against doing this because configuring a build script with Android dependencies in the top-level block and other target dependencies in each source set is likely to cause confusion.

What's next?

Check out other resources on adding dependencies in multiplatform projects and learn more about:

- Adding dependencies in the official Android documentation
- Adding dependencies on multiplatform libraries or other multiplatform projects
- Adding iOS dependencies

Adding iOS dependencies

Apple SDK dependencies (such as Foundation or Core Bluetooth) are available as a set of prebuilt libraries in Kotlin Multiplatform Mobile projects. They do not require any additional configuration.

You can also reuse other libraries and frameworks from the iOS ecosystem in your iOS source sets. Kotlin supports interoperability with Objective-C dependencies and Swift dependencies if their APIs are exported to Objective-C with the @objc attribute. Pure Swift dependencies are not yet supported.

Integration with the CocoaPods dependency manager is also supported with the same limitation – you cannot use pure Swift pods.

We recommend using CocoaPods to handle iOS dependencies in Kotlin Multiplatform projects. Manage dependencies manually only if you want to tune the interop process specifically or if you have some other strong reason to do so.

With CocoaPods

1. Perform initial CocoaPods integration setup.
2. Add a dependency on a Pod library from the CocoaPods repository that you want to use by including the pod() function call in build.gradle(.kts) of your project.

   Kotlin

   ```
   kotlin {
       cocoapods {
           //..
           pod("AFNetworking") {
               version = "~> 4.0.1"
           }
       }
   }
   ```

   Groovy

   ```
   kotlin {
       cocoapods {
           //..
           pod('AFNetworking') {
               version = '~> 4.0.1'
           }
       }
   }
   ```

   You can add the following dependencies on a Pod library:

   - From the CocoaPods repository

- On a locally stored library
- From a custom Git repository
- From a custom Podspec repository
- With custom cinterop options

3. Re-import the project.

To use the dependency in your Kotlin code, import the package cocoapods.<library-name>. For the example above, it's:

```
import cocoapods.AFNetworking.*
```

Without CocoaPods

If you don't want to use CocoaPods, you can use the cinterop tool to create Kotlin bindings for Objective-C or Swift declarations. This will allow you to call them from the Kotlin code.

The steps differ a bit for libraries and frameworks, but the idea remains the same.

1. Download your dependency.
2. Build it to get its binaries.
3. Create a special .def file that describes this dependency to cinterop.
4. Adjust your build script to generate bindings during the build.

Add a library without CocoaPods

1. Download the library source code and place it somewhere where you can reference it from your project.
2. Build a library (library authors usually provide a guide on how to do this) and get a path to the binaries.
3. In your project, create a .def file, for example DateTools.def.
4. Add a first string to this file: language = Objective-C. If you want to use a pure C dependency, omit the language property.
5. Provide values for two mandatory properties:
   - headers describes which headers will be processed by cinterop.
   - package sets the name of the package these declarations should be put into.

   For example:

   ```
   headers = DateTools.h
   package = DateTools
   ```

6. Add information about interoperability with this library to the build script:
   - Pass the path to the .def file. This path can be omitted if your .def file has the same name as cinterop and is placed in the src/nativeInterop/cinterop/ directory.
   - Tell cinterop where to look for header files using the includeDirs option.
   - Configure linking to library binaries.

   Kotlin

   ```
   kotlin {
       iosX64() {
           compilations.getByName("main") {
               val DateTools by cinterops.creating {
                   // Path to .def file
                   defFile("src/nativeInterop/cinterop/DateTools.def")
   ```

```
                // Directories for header search (an analogue of the -I<path> compiler option)
                includeDirs("include/this/directory", "path/to/another/directory")
            }
            val anotherInterop by cinterops.creating { /* ... */ }
        }

        binaries.all {
            // Linker options required to link to the library.
            linkerOpts("-L/path/to/library/binaries", "-lbinaryname")
        }
    }
}
```

Groovy

```
kotlin {
    iosX64 {
        compilations.main {
            cinterops {
                DateTools {
                    // Path to .def file
                    defFile("src/nativeInterop/cinterop/DateTools.def")

                    // Directories for header search (an analogue of the -I<path> compiler option)
                    includeDirs("include/this/directory", "path/to/another/directory")
                }
                anotherInterop { /* ... */ }
            }
        }

        binaries.all {
            // Linker options required to link to the library.
            linkerOpts "-L/path/to/library/binaries", "-lbinaryname"
        }
    }
}
```

7. Build the project.

Now you can use this dependency in your Kotlin code. To do that, import the package you've set up in the package property in the .def file. For the example above, this will be:

```
import DateTools.*
```

Add a framework without CocoaPods

1. Download the framework source code and place it somewhere that you can reference it from your project.
2. Build the framework (framework authors usually provide a guide on how to do this) and get a path to the binaries.
3. In your project, create a .def file, for example MyFramework.def.
4. Add the first string to this file: language = Objective-C. If you want to use a pure C dependency, omit the language property.
5. Provide values for these two mandatory properties:
   - modules – the name of the framework that should be processed by the cinterop.
   - package – the name of the package these declarations should be put into.

   For example:

   ```
   modules = MyFramework
   package = MyFramework
   ```
6. Add information about interoperability with the framework to the build script:

- Pass the path to the .def file. This path can be omitted if your .def file has the same name as the cinterop and is placed in the src/nativeInterop/cinterop/ directory.

- Pass the framework name to the compiler and linker using the -framework option. Pass the path to the framework sources and binaries to the compiler and linker using the -F option.

Kotlin

```kotlin
kotlin {
    iosX64() {
        compilations.getByName("main") {
            val DateTools by cinterops.creating {
                // Path to .def file
                defFile("src/nativeInterop/cinterop/DateTools.def")

                compilerOpts("-framework", "MyFramework", "-F/path/to/framework/")
            }
            val anotherInterop by cinterops.creating { /* ... */ }
        }

        binaries.all {
            // Tell the linker where the framework is located.
            linkerOpts("-framework", "MyFramework", "-F/path/to/framework/")
        }
    }
}
```

Groovy

```groovy
kotlin {
    iosX64 {
        compilations.main {
            cinterops {
                DateTools {
                    // Path to .def file
                    defFile("src/nativeInterop/cinterop/MyFramework.def")

                    compilerOpts("-framework", "MyFramework", "-F/path/to/framework/")
                }
                anotherInterop { /* ... */ }
            }
        }

        binaries.all {
            // Tell the linker where the framework is located.
            linkerOpts("-framework", "MyFramework", "-F/path/to/framework/")
        }
    }
}
```

7. Build the project.

Now you can use this dependency in your Kotlin code. To do this, import the package you've set up in the package property in the .def file. For the example above, this will be:

```kotlin
import MyFramework.*
```

Learn more about Objective-C and Swift interop and configuring cinterop from Gradle.

What's next?

Check out other resources on adding dependencies in multiplatform projects and learn more about:

- Connecting platform-specific libraries
- Adding dependencies on multiplatform libraries or other multiplatform projects
- Adding Android dependencies

Run tests with Kotlin Multiplatform

By default, Kotlin supports running tests for JVM, JS, Android, Linux, Windows, macOS as well as iOS, watchOS, and tvOS simulators. To run tests for other Kotlin/Native targets, you need to configure them manually in an appropriate environment, emulator, or test framework.

Required dependencies

The kotlin.test API is available for multiplatform tests. When you create a multiplatform project, the Project Wizard automatically adds test dependencies to common and platform-specific source sets.

If you didn't use the Project Wizard to create your project, you can add the dependencies manually.

Run tests for one or more targets

To run tests for all targets, run the check task.

To run tests for a particular target suitable for testing, run a test task <targetName>Test.

Test shared code

For testing shared code, you can use actual declarations in your tests.

For example, to test the shared code in commonMain:

```
expect object Platform {
    val name: String
}

fun hello(): String = "Hello from ${Platform.name}"

class Proxy {
    fun proxyHello() = hello()
}
```

You can use the following test in commonTest:

```
import kotlin.test.Test
import kotlin.test.assertTrue

class SampleTests {
    @Test
    fun testProxy() {
        assertTrue(Proxy().proxyHello().isNotEmpty())
    }
}
```

And the following test in iosTest:

```
import kotlin.test.Test
import kotlin.test.assertTrue

class SampleTestsIOS {
    @Test
    fun testHello() {
        assertTrue("iOS" in hello())
    }
}
```

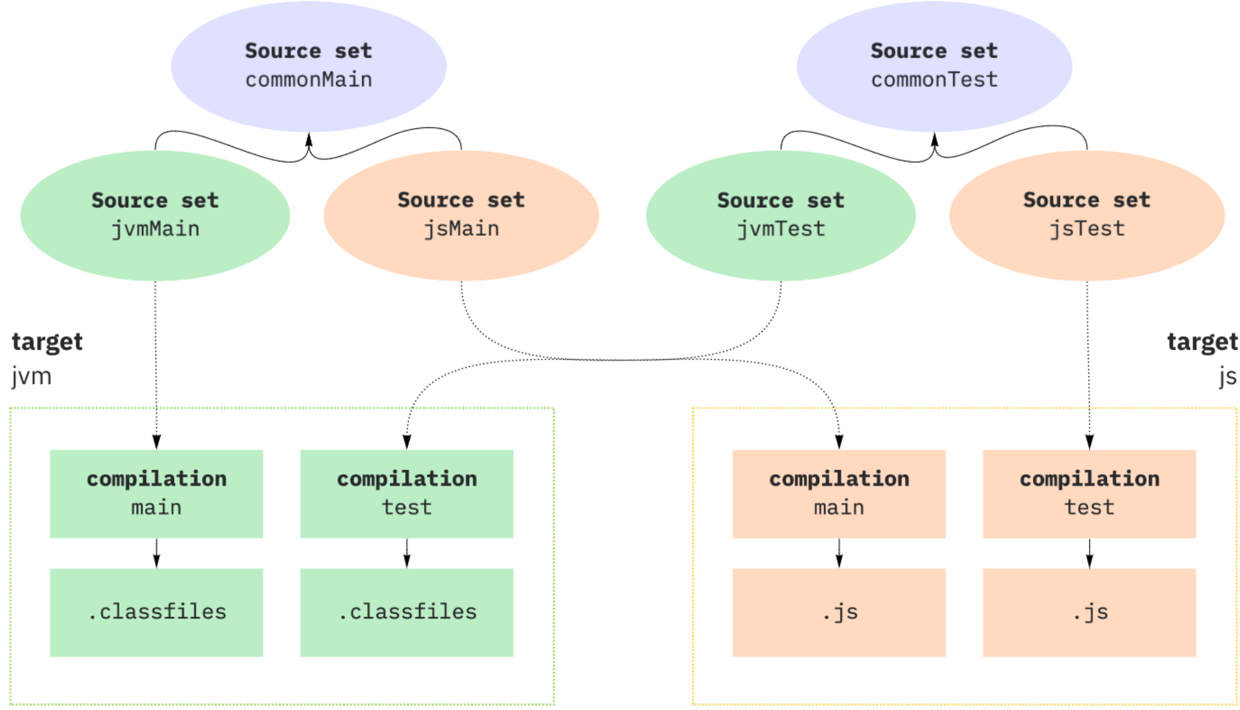
You can also learn how to create and run multiplatform tests in the Create and publish a multiplatform library – tutorial.

Configure compilations

Kotlin multiplatform projects use compilations for producing artifacts. Each target can have one or more compilations, for example, for production and test purposes.

For each target, default compilations include:

- main and test compilations for JVM, JS, and Native targets.
- A compilation per Android build variant, for Android targets.



Compilations

If you need to compile something other than production code and unit tests, for example, integration or performance tests, you can create a custom compilation.

You can configure how artifacts are produced in:

- All compilations in your project at once.
- Compilations for one target since one target can have multiple compilations.
- A specific compilation.

See the list of compilation parameters and compiler options available for all or specific targets.

Configure all compilations

```
kotlin {
    targets.all {
        compilations.all {
            compilerOptions.configure {
                allWarningsAsErrors.set(true)
            }
        }
    }
}
```

Configure compilations for one target

Kotlin

```kotlin
kotlin {
    targets.jvm.compilations.all {
        compilerOptions.configure {
            sourceMap.set(true)
            metaInfo.set(true)
        }
    }
}
```

Groovy

```groovy
kotlin {
    jvm().compilations.all {
        compilerOptions.configure {
            sourceMap.set(true)
            metaInfo.set(true)
        }
    }
}
```

Configure one compilation

Kotlin

```kotlin
kotlin {
    jvm {
        val main by compilations.getting {
            compilerOptions.configure {
                jvmTarget.set(JvmTarget.JVM_1_8)
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    jvm().compilations.main {
        compilerOptions.configure {
            jvmTarget.set(JvmTarget.JVM_1_8)
        }
    }
}
```

Create a custom compilation

If you need to compile something other than production code and unit tests, for example, integration or performance tests, create a custom compilation.

For example, to create a custom compilation for integration tests of the jvm() target, add a new item to the compilations collection.

> For custom compilations, you need to set up all dependencies manually. The default source set of a custom compilation does not depend on the commonMain and the commonTest source sets.

Kotlin

```kotlin
kotlin {
    jvm() {
        compilations {
            val main by getting

            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
```

```
                    // Compile against the main compilation's compile classpath and outputs:
                    implementation(main.compileDependencyFiles + main.output.classesDirs)
                    implementation(kotlin("test-junit"))
                    /* ... */
                }
            }

            // Create a test task to run the tests produced by this compilation:
            tasks.register<Test>("integrationTest") {
                // Run the tests with the classpath containing the compile dependencies (including 'main'),
                // runtime dependencies, and the outputs of this compilation:
                classpath = compileDependencyFiles + runtimeDependencyFiles + output.allOutputs

                // Run only the tests from this compilation's outputs:
                testClassesDirs = output.classesDirs
            }
        }
    }
}
```

Groovy

```
kotlin {
    jvm() {
        compilations.create('integrationTest') {
            defaultSourceSet {
                dependencies {
                    def main = compilations.main
                    // Compile against the main compilation's compile classpath and outputs:
                    implementation(main.compileDependencyFiles + main.output.classesDirs)
                    implementation kotlin('test-junit')
                    /* ... */
                }
            }

            // Create a test task to run the tests produced by this compilation:
            tasks.register('jvmIntegrationTest', Test) {
                // Run the tests with the classpath containing the compile dependencies (including 'main'),
                // runtime dependencies, and the outputs of this compilation:
                classpath = compileDependencyFiles + runtimeDependencyFiles + output.allOutputs

                // Run only the tests from this compilation's outputs:
                testClassesDirs = output.classesDirs
            }
        }
    }
}
```

You also need to create a custom compilation in other cases, for example, if you want to combine compilations for different JVM versions in your final artifact, or you have already set up source sets in Gradle and want to migrate to a multiplatform project.

Use Java sources in JVM compilations

When creating a project with the Project Wizard, Java sources are included in the compilations of the JVM target.

In the build script, the following section applies the Gradle java plugin and configures the target to cooperate with it:

```
kotlin {
    jvm {
        withJava()
    }
}
```

The Java source files are placed in the child directories of the Kotlin source roots. For example, the paths are:

Java source files

The common source sets cannot include Java sources.

Due to current limitations, the Kotlin plugin replaces some tasks configured by the Java plugin:

- The target's JAR task instead of jar (for example, jvmJar).
- The target's test task instead of test (for example, jvmTest).
- The resources are processed by the equivalent tasks of the compilations instead of *ProcessResources tasks.

The publication of this target is handled by the Kotlin plugin and doesn't require steps that are specific for the Java plugin.

Configure interop with native languages

Kotlin provides interoperability with native languages and DSL to configure this for a specific compilation.

Native language	Supported platforms	Comments
C	All platforms, except for WebAssembly	
Objective-C	Apple platforms (macOS, iOS, watchOS, tvOS)	
Swift via Objective-C	Apple platforms (macOS, iOS, watchOS, tvOS)	Kotlin can use only Swift declarations marked with the @objc attribute.

A compilation can interact with several native libraries. Configure interoperability in the cinterops block of the compilation with available parameters.

Kotlin

```
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.getByName("main") {
            val myInterop by cinterops.creating {
                // Def-file describing the native API.
                // The default path is src/nativeInterop/cinterop/<interop-name>.def
                defFile(project.file("def-file.def"))

                // Package to place the Kotlin API generated.
```

```
                packageName("org.sample")

                // Options to be passed to compiler by cinterop tool.
                compilerOpts("-Ipath/to/headers")

                // Directories to look for headers.
                includeDirs.apply {
                    // Directories for header search (an equivalent of the -I<path> compiler option).
                    allHeaders("path1", "path2")

                    // Additional directories to search headers listed in the 'headerFilter' def-file option.
                    // -headerFilterAdditionalSearchPrefix command line option equivalent.
                    headerFilterOnly("path1", "path2")
                }
                // A shortcut for includeDirs.allHeaders.
                includeDirs("include/directory", "another/directory")
            }

            val anotherInterop by cinterops.creating { /* ... */ }
        }
    }
}
```

Groovy

```
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.main {
            cinterops {
                myInterop {
                    // Def-file describing the native API.
                    // The default path is src/nativeInterop/cinterop/<interop-name>.def
                    defFile project.file("def-file.def")

                    // Package to place the Kotlin API generated.
                    packageName 'org.sample'

                    // Options to be passed to compiler by cinterop tool.
                    compilerOpts '-Ipath/to/headers'

                    // Directories for header search (an eqivalent of the -I<path> compiler option).
                    includeDirs.allHeaders("path1", "path2")

                    // Additional directories to search headers listed in the 'headerFilter' def-file option.
                    // -headerFilterAdditionalSearchPrefix command line option equivalent.
                    includeDirs.headerFilterOnly("path1", "path2")

                    // A shortcut for includeDirs.allHeaders.
                    includeDirs("include/directory", "another/directory")
                }

                anotherInterop { /* ... */ }
            }
        }
    }
}
```

Compilation for Android

The compilations created for an Android target by default are tied to Android build variants: for each build variant, a Kotlin compilation is created under the same name.

Then, for each Android source set compiled for each of the variants, a Kotlin source set is created under that source set name prepended by the target name, like the Kotlin source set androidDebug for an Android source set debug and the Kotlin target named android. These Kotlin source sets are added to the variants' compilations accordingly.

The default source set commonMain is added to each production (application or library) variant's compilation. The commonTest source set is similarly added to the compilations of unit test and instrumented test variants.

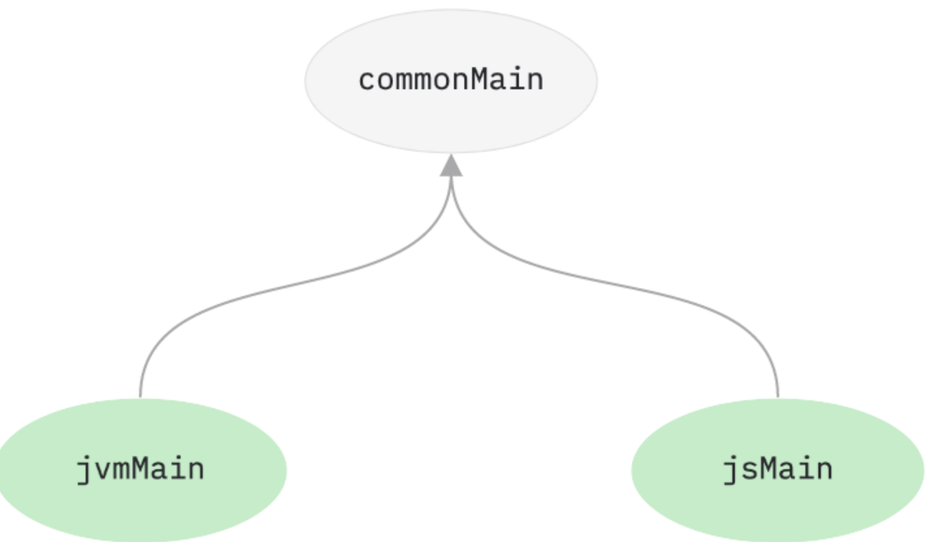
Annotation processing with kapt is also supported, but due to current limitations it requires that the Android target is created before the kapt dependencies are configured, which needs to be done in a top-level dependencies block rather than within Kotlin source set dependencies.

```
kotlin {
    android { /* ... */ }
```

```
}

dependencies {
    kapt("com.my.annotation:processor:1.0.0")
}
```

Compilation of the source set hierarchy

Kotlin can build a source set hierarchy with the dependsOn relation.



Source set hierarchy

If the source set jvmMain depends on a source set commonMain then:

- Whenever jvmMain is compiled for a certain target, commonMain takes part in that compilation as well and is also compiled into the same target binary form, such as JVM class files.
- Sources of jvmMain 'see' the declarations of commonMain, including internal declarations, and also see the dependencies of commonMain, even those specified as implementation dependencies.
- jvmMain can contain platform-specific implementations for the expected declarations of commonMain.
- The resources of commonMain are always processed and copied along with the resources of jvmMain.
- The language settings of jvmMain and commonMain should be consistent.

Language settings are checked for consistency in the following ways:

- jvmMain should set a languageVersion that is greater than or equal to that of commonMain.
- jvmMain should enable all unstable language features that commonMain enables (there's no such requirement for bugfix features).
- jvmMain should use all experimental annotations that commonMain uses.
- apiVersion, bugfix language features, and progressiveMode can be set arbitrarily.

Build final native binaries (Experimental DSL)

> The new DSL described below is Experimental. It may be changed at any time. We encourage you to use it for evaluation purposes.
>
> If the new DSL doesn't work for you, see the previous approach to building native binaries.

Kotlin/Native targets are compiled to the *.klib library artifacts, which can be consumed by Kotlin/Native itself as a dependency but cannot be used as a native library.

To declare final native binaries, use the new binaries format with the kotlinArtifacts DSL. It represents a collection of native binaries built for this target in addition to the default *.klib artifact and provides a set of methods for declaring and configuring them.

> The kotlin-multiplatform plugin doesn't create any production binaries by default. The only binary available by default is a debug test executable that lets you run unit tests from the test compilation.

Kotlin artifact DSL can help you to solve a common issue: when you need to access multiple Kotlin modules from your app. Since the usage of several Kotlin/Native artifacts is limited, you can export multiple Kotlin modules into a single artifact with new DSL.

Declare binaries

The kotlinArtifacts element is the top-level block for artifact configuration in the Gradle build script. Use the following kinds of binaries to declare elements of the kotlinArtifacts DSL:

Factory method	Binary kind	Available for
sharedLib	Shared native library	All native targets, except for WebAssembly
staticLib	Static native library	All native targets, except for WebAssembly
framework	Objective-C framework	macOS, iOS, watchOS, and tvOS targets only
fatFramework	Universal fat framework	macOS, iOS, watchOS, and tvOS targets only
XCFramework	XCFramework framework	macOS, iOS, watchOS, and tvOS targets only

Inside the kotlinArtifacts element, you can write the following blocks:

- Native.Library
- Native.Framework
- Native.FatFramework
- Native.XCFramework

The simplest version requires the target (or targets) parameter for the selected build type. Currently, two build types are available:

- DEBUG – produces a non-optimized binary with debug information
- RELEASE – produces an optimized binary without debug information

In the modes parameter, you can specify build types for which you want to create binaries. The default value includes both DEBUG and RELEASE executable binaries:

Kotlin

```
kotlinArtifacts {
    Native.Library {
        target = iosX64 // Define your target instead
        modes(DEBUG, RELEASE)
        // Binary configuration
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.Library {
        target = iosX64 // Define your target instead
        modes(DEBUG, RELEASE)
        // Binary configuration
    }
}
```

You can also declare binaries with custom names:

Kotlin

```
kotlinArtifacts {
    Native.Library("mylib") {
        // Binary configuration
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.Library("mylib") {
        // Binary configuration
    }
}
```

The argument sets a name prefix, which is the default name for the binary file. For example, for Windows the code produces the mylib.dll file.

Configure binaries

For the binary configuration, the following common parameters are available:

Name	Description
isStatic	Optional linking type that defines the library type. By default, it's false and the library is dynamic.
modes	Optional build types, DEBUG and RELEASE.
kotlinOptions	Optional compiler options applied to the compilation. See the list of available compiler options.
addModule	In addition to the current module, you can add other modules to the resulting artifact.
setModules	You can override the list of all modules that will be added to the resulting artifact.

Libraries and frameworks

When building an Objective-C framework or a native library (shared or static), you may need to pack not just the classes of the current project but also the classes of any other multiplatform module into a single entity and export all these modules to it.

Library

For the library configuration, the additional target parameter is available:

Name	Description
target	Declares a particular target of a project. The names of available targets are listed in the Targets section.

Kotlin

```kotlin
kotlinArtifacts {
    Native.Library("myslib") {
        target = linuxX64
        isStatic = false
        modes(DEBUG)
        addModule(project(":lib"))
        kotlinOptions {
            verbose = false
            freeCompilerArgs += "-Xmen=pool"
        }
    }
}
```

Groovy

```groovy
kotlinArtifacts {
    it.native.Library("myslib") {
        target = linuxX64
        it.static = false
        modes(DEBUG)
        addModule(project(":lib"))
        kotlinOptions {
            verbose = false
            freeCompilerArgs += "-Xmen=pool"
        }
    }
}
```

The registered Gradle task is assembleMyslibSharedLibrary that assembles all types of registered "myslib" into a dynamic library.

Framework

For the framework configuration, the following additional parameters are available:

Name	Description
target	Declares a particular target of a project. The names of available targets are listed in the Targets section.
embedBitcode	Declares the mode of bitcode embedding. Use MARKER to embed the bitcode marker (for debug builds) or DISABLE to turn off embedding. Bitcode embedding is not required for Xcode 14 and later.

Kotlin

```kotlin
kotlinArtifacts {
    Native.Framework("myframe") {
```

```
        modes(DEBUG, RELEASE)
        target = iosArm64
        isStatic = false
        embedBitcode = EmbedBitcodeMode.MARKER
        kotlinOptions {
            verbose = false
        }
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.Framework("myframe") {
        modes(DEBUG, RELEASE)
        target = iosArm64
        it.static = false
        embedBitcode = EmbedBitcodeMode.MARKER
        kotlinOptions {
            verbose = false
        }
    }
}
```

The registered Gradle task is assembleMyframeFramework that assembles all types of registered "myframe" framework.

> If for some reason the new DSL doesn't work for you, try the previous approach to export dependencies to binaries.

Fat frameworks

By default, an Objective-C framework produced by Kotlin/Native supports only one platform. However, you can merge such frameworks into a single universal (fat) binary. This especially makes sense for 32-bit and 64-bit iOS frameworks. In this case, you can use the resulting universal framework on both 32-bit and 64-bit devices.

For the fat framework configuration, the following additional parameters are available:

Name	Description
targets	Declares all targets of the project.
embedBitcode	Declares the mode of bitcode embedding. Use MARKER to embed the bitcode marker (for debug builds) or DISABLE to turn off embedding. Bitcode embedding is not required for Xcode 14 and later.

Kotlin

```
kotlinArtifacts {
    Native.FatFramework("myfatframe") {
        targets(iosX32, iosX64)
        embedBitcode = EmbedBitcodeMode.DISABLE
        kotlinOptions {
            suppressWarnings = false
        }
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.FatFramework("myfatframe") {
        targets(iosX32, iosX64)
        embedBitcode = EmbedBitcodeMode.DISABLE
        kotlinOptions {
```

```
            suppressWarnings = false
        }
    }
}
```

The registered Gradle task is assembleMyfatframeFatFramework that assembles all types of registered "myfatframe" fat framework.

> If for some reason the new DSL doesn't work for you, try the previous approach to build fat frameworks.

XCFrameworks

All Kotlin Multiplatform projects can use XCFrameworks as an output to gather logic for all the target platforms and architectures in a single bundle. Unlike universal (fat) frameworks, you don't need to remove all unnecessary architectures before publishing the application to the App Store.

For the XCFrameworks configuration, the following additional parameters are available:

Name	Description
targets	Declares all targets of the project.
embedBitcode	Declares the mode of bitcode embedding. Use MARKER to embed the bitcode marker (for debug builds) or DISABLE to turn off embedding. Bitcode embedding is not required for Xcode 14 and later.

Kotlin

```kotlin
kotlinArtifacts {
    Native.XCFramework("sdk") {
        targets(iosX64, iosArm64, iosSimulatorArm64)
        setModules(
            project(":shared"),
            project(":lib")
        )
    }
}
```

Groovy

```groovy
kotlinArtifacts {
    it.native.XCFramework("sdk") {
        targets(iosX64, iosArm64, iosSimulatorArm64)
        setModules(
            project(":shared"),
            project(":lib")
        )
    }
}
```

The registered Gradle task is assembleSdkXCFramework that assembles all types of registered "sdk" XCFrameworks.

> If for some reason the new DSL doesn't work for you, try the previous approach to build XCFrameworks.

Build final native binaries

By default, a Kotlin/Native target is compiled down to a *.klib library artifact, which can be consumed by Kotlin/Native itself as a dependency but cannot be executed or used as a native library.

To declare final native binaries such as executables or shared libraries, use the binaries property of a native target. This property represents a collection of native binaries built for this target in addition to the default *.klib artifact and provides a set of methods for declaring and configuring them.

> The kotlin-multiplatform plugin doesn't create any production binaries by default. The only binary available by default is a debug test executable that lets you run unit tests from the test compilation.

Binaries produced by the Kotlin/Native compiler can include third-party code, data, or derived work. This means if you distribute a Kotlin/Native-compiled final binary, you should always include necessary license files into your binary distribution.

Declare binaries

Use the following factory methods to declare elements of the binaries collection.

Factory method	Binary kind	Available for
executable	Product executable	All native targets
test	Test executable	All native targets
sharedLib	Shared native library	All native targets, except for WebAssembly
staticLib	Static native library	All native targets, except for WebAssembly
framework	Objective-C framework	macOS, iOS, watchOS, and tvOS targets only

The simplest version doesn't require any additional parameters and creates one binary for each build type. Currently, two build types are available:

- DEBUG – produces a non-optimized binary with debug information
- RELEASE – produces an optimized binary without debug information

The following snippet creates two executable binaries, debug and release:

```
kotlin {
    linuxX64 { // Define your target instead.
        binaries {
            executable {
                // Binary configuration.
            }
        }
    }
}
```

You can drop the lambda if there is no need for additional configuration:

```
binaries {
    executable()
}
```

You can specify for which build types to create binaries. In the following example, only the debug executable is created:

Kotlin

```
binaries {
    executable(listOf(DEBUG)) {
```

```
        // Binary configuration.
    }
}
```

Groovy

```
binaries {
    executable([DEBUG]) {
        // Binary configuration.
    }
}
```

You can also declare binaries with custom names:

Kotlin

```
binaries {
    executable("foo", listOf(DEBUG)) {
        // Binary configuration.
    }

    // It's possible to drop the list of build types
    // (in this case, all the available build types will be used).
    executable("bar") {
        // Binary configuration.
    }
}
```

Groovy

```
binaries {
    executable('foo', [DEBUG]) {
        // Binary configuration.
    }

    // It's possible to drop the list of build types
    // (in this case, all the available build types will be used).
    executable('bar') {
        // Binary configuration.
    }
}
```

The first argument sets a name prefix, which is the default name for the binary file. For example, for Windows the code produces the files foo.exe and bar.exe. You can also use the name prefix to access the binary in the build script.

Access binaries

You can access binaries to configure them or get their properties (for example, the path to an output file).

You can get a binary by its unique name. This name is based on the name prefix (if it is specified), build type, and binary kind following the pattern: <optional-name-prefix><build-type><binary-kind>, for example, releaseFramework or testDebugExecutable.

> Static and shared libraries have the suffixes static and shared respectively, for example, fooDebugStatic or barReleaseShared.

Kotlin

```
// Fails if there is no such binary.
binaries["fooDebugExecutable"]
binaries.getByName("fooDebugExecutable")

// Returns null if there is no such binary.
binaries.findByName("fooDebugExecutable")
```

Groovy

```groovy
// Fails if there is no such binary.
binaries['fooDebugExecutable']
binaries.fooDebugExecutable
binaries.getByName('fooDebugExecutable')

// Returns null if there is no such binary.
binaries.findByName('fooDebugExecutable')
```

Alternatively, you can access a binary by its name prefix and build type using typed getters.

Kotlin

```kotlin
// Fails if there is no such binary.
binaries.getExecutable("foo", DEBUG)
binaries.getExecutable(DEBUG)          // Skip the first argument if the name prefix isn't set.
binaries.getExecutable("bar", "DEBUG") // You also can use a string for build type.

// Similar getters are available for other binary kinds:
// getFramework, getStaticLib and getSharedLib.

// Returns null if there is no such binary.
binaries.findExecutable("foo", DEBUG)

// Similar getters are available for other binary kinds:
// findFramework, findStaticLib and findSharedLib.
```

Groovy

```groovy
// Fails if there is no such binary.
binaries.getExecutable('foo', DEBUG)
binaries.getExecutable(DEBUG)          // Skip the first argument if the name prefix isn't set.
binaries.getExecutable('bar', 'DEBUG') // You also can use a string for build type.

// Similar getters are available for other binary kinds:
// getFramework, getStaticLib and getSharedLib.

// Returns null if there is no such binary.
binaries.findExecutable('foo', DEBUG)

// Similar getters are available for other binary kinds:
// findFramework, findStaticLib and findSharedLib.
```

Export dependencies to binaries

When building an Objective-C framework or a native library (shared or static), you may need to pack not just the classes of the current project, but also the classes of its dependencies. Specify which dependencies to export to a binary using the export method.

Kotlin

```kotlin
kotlin {
    sourceSets {
        macosMain.dependencies {
            // Will be exported.
            api(project(":dependency"))
            api("org.example:exported-library:1.0")
            // Will not be exported.
            api("org.example:not-exported-library:1.0")
        }
    }
    macosX64("macos").binaries {
        framework {
            export(project(":dependency"))
            export("org.example:exported-library:1.0")
        }
        sharedLib {
            // It's possible to export different sets of dependencies to different binaries.
            export(project(':dependency'))
```

```
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // Will be exported.
            api project(':dependency')
            api 'org.example:exported-library:1.0'
            // Will not be exported.
            api 'org.example:not-exported-library:1.0'
        }
    }
    macosX64("macos").binaries {
        framework {
            export project(':dependency')
            export 'org.example:exported-library:1.0'
        }
        sharedLib {
            // It's possible to export different sets of dependencies to different binaries.
            export project(':dependency')
        }
    }
}
```

For example, you implement several modules in Kotlin and want to access them from Swift. Usage of several Kotlin/Native frameworks in a Swift application is limited, but you can create an umbrella framework and export all these modules to it.

> You can export only api dependencies of the corresponding source set.

When you export a dependency, it includes all of its API to the framework API. The compiler adds the code from this dependency to the framework, even if you use a small fraction of it. This disables dead code elimination for the exported dependency (and for its dependencies, to some extent).

By default, export works non-transitively. This means that if you export the library foo depending on the library bar, only methods of foo are added to the output framework.

You can change this behavior using the transitiveExport option. If set to true, the declarations of the library bar are exported as well.

> It is not recommended to use transitiveExport: it adds all transitive dependencies of the exported dependencies to the framework. This could increase both compilation time and binary size.
>
> In most cases, you don't need to add all these dependencies to the framework API. Use export explicitly for the dependencies you need to directly access from your Swift or Objective-C code.

Kotlin

```
binaries {
    framework {
        export(project(":dependency"))
        // Export transitively.
        transitiveExport = true
    }
}
```

Groovy

```
binaries {
    framework {
        export project(':dependency')
        // Export transitively.
        transitiveExport = true
    }
}
```

Build universal frameworks

By default, an Objective-C framework produced by Kotlin/Native supports only one platform. However, you can merge such frameworks into a single universal (fat) binary using the lipo tool. This operation especially makes sense for 32-bit and 64-bit iOS frameworks. In this case, you can use the resulting universal framework on both 32-bit and 64-bit devices.

> The fat framework must have the same base name as the initial frameworks. Otherwise, you'll get an error.

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // Create and configure the targets.
    val ios32 = watchosArm32("watchos32")
    val ios64 = watchosArm64("watchos64")
    configure(listOf(watchos32, watchos64)) {
        binaries.framework {
            baseName = "my_framework"
        }
    }
    // Create a task to build a fat framework.
    tasks.register<FatFrameworkTask>("debugFatFramework") {
        // The fat framework must have the same base name as the initial frameworks.
        baseName = "my_framework"
        // The default destination directory is "<build directory>/fat-framework".
        destinationDir = buildDir.resolve("fat-framework/debug")
        // Specify the frameworks to be merged.
        from(
            ios32.binaries.getFramework("DEBUG"),
            ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // Create and configure the targets.
    targets {
        watchosArm32("watchos32")
        watchosArm64("watchos64")
        configure([watchos32, watchos64]) {
            binaries.framework {
                baseName = "my_framework"
            }
        }
    }
    // Create a task building a fat framework.
    tasks.register("debugFatFramework", FatFrameworkTask) {
        // The fat framework must have the same base name as the initial frameworks.
        baseName = "my_framework"
        // The default destination directory is "<build directory>/fat-framework".
        destinationDir = file("$buildDir/fat-framework/debug")
        // Specify the frameworks to be merged.
        from(
            targets.ios32.binaries.getFramework("DEBUG"),
            targets.ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

Build XCFrameworks

All Kotlin Multiplatform projects can use XCFrameworks as an output to gather logic for all the target platforms and architectures in a single bundle. Unlike universal

(fat) frameworks, you don't need to remove all unnecessary architectures before publishing the application to the App Store.

Kotlin

```
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFramework

plugins {
    kotlin("multiplatform")
}

kotlin {
    val xcf = XCFramework()
    val iosTargets = listOf(iosX64(), iosArm64(), iosSimulatorArm64())

    iosTargets.forEach {
        it.binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFrameworkConfig

plugins {
    id 'org.jetbrains.kotlin.multiplatform'
}

kotlin {
    def xcf = new XCFrameworkConfig(project)
    def iosTargets = [iosX64(), iosArm64(), iosSimulatorArm64()]

    iosTargets.forEach {
        it.binaries.framework {
            baseName = 'shared'
            xcf.add(it)
        }
    }
}
```

When you declare XCFrameworks, Kotlin Gradle plugin will register three Gradle tasks:

- assembleXCFramework
- assembleDebugXCFramework (additionally debug artifact that contains dSYMs)
- assembleReleaseXCFramework

If you're using CocoaPods integration in your projects, you can build XCFrameworks with the Kotlin CocoaPods Gradle plugin. It includes the following tasks that build XCFrameworks with all the registered targets and generate podspec files:

- podPublishReleaseXCFramework, which generates a release XCFramework along with a podspec file.
- podPublishDebugXCFramework, which generates a debug XCFramework along with a podspec file.
- podPublishXCFramework, which generates both debug and release XCFrameworks along with a podspec file.

This can help you distribute shared parts of your project separately from mobile apps through CocoaPods. You can also use XCFrameworks for publishing to private or public podspec repositories.

> Publishing Kotlin frameworks to public repositories is not recommended if those frameworks are built for different versions of Kotlin. Doing so might lead to conflicts in the end-users' projects.

Customize the Info.plist file

When producing a framework, the Kotlin/Native compiler generates the information property list file, Info.plist. You can customize its properties with the

corresponding binary option:

Property	Binary option
CFBundleIdentifier	bundleId
CFBundleShortVersionString	bundleShortVersionString
CFBundleVersion	bundleVersion

To enable the feature, pass the -Xbinary=$option=$value compiler flag or set the binaryOption("option", "value") Gradle DSL for the specific framework:

```
binaries {
    framework {
        binaryOption("bundleId", "com.example.app")
        binaryOption("bundleVersion", "2")
    }
}
```

Multiplatform Gradle DSL reference

Multiplatform projects are in Alpha. Language features and tooling may change in future Kotlin versions.

The Kotlin Multiplatform Gradle plugin is a tool for creating Kotlin Multiplatform projects. Here we provide a reference of its contents; use it as a reminder when writing Gradle build scripts for Kotlin Multiplatform projects. Learn the concepts of Kotlin Multiplatform projects, how to create and configure them.

Id and version

The fully qualified name of the Kotlin Multiplatform Gradle plugin is org.jetbrains.kotlin.multiplatform. If you use the Kotlin Gradle DSL, you can apply the plugin with kotlin("multiplatform"). The plugin versions match the Kotlin release versions. The most recent version is 1.9.0.

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}
```

Top-level blocks

kotlin is the top-level block for multiplatform project configuration in the Gradle build script. Inside kotlin, you can write the following blocks:

Block	Description

Block	Description
<targetName>	Declares a particular target of a project. The names of available targets are listed in the Targets section.
targets	All targets of the project.
presets	All predefined targets. Use this for configuring multiple predefined targets at once.
sourceSets	Configures predefined and declares custom source sets of the project.

Targets

Target is a part of the build responsible for compiling, testing, and packaging a piece of software aimed at one of the supported platforms. Kotlin provides target presets for each platform. See how to use a target preset.

Each target can have one or more compilations. In addition to default compilations for test and production purposes, you can create custom compilations.

The targets of a multiplatform project are described in the corresponding blocks inside kotlin, for example, jvm, android, iosArm64. The complete list of available targets is the following:

Target platform	Target preset	Comments
Kotlin/JVM	jvm	
Kotlin/JS	js	Select the execution environment: • browser {} for applications running in the browser. • nodejs {} for applications running on Node.js. Learn more in Setting up a Kotlin/JS project.
Kotlin/Native		Learn about currently supported targets for the macOS, Linux, and Windows hosts in Kotlin/Native target support.
Android applications and libraries	android	Manually apply an Android Gradle plugin: com.android.application or com.android.library. You can only create one Android target per Gradle subproject.

A target that is not supported by the current host is ignored during building and, therefore, not published.

```
kotlin {
    jvm()
    iosX64()
    macosX64()
    js().browser()
}
```

The configuration of a target can include two parts:

- Common configuration available for all targets.
- Target-specific configuration.

Each target can have one or more compilations.

Common target configuration

In any target block, you can use the following declarations:

Name	Description
attributes	Attributes used for disambiguating targets for a single platform.
preset	The preset that the target has been created from, if any.
platformType	Designates the Kotlin platform of this target. Available values: jvm, androidJvm, js, native, common.
artifactsTaskName	The name of the task that builds the resulting artifacts of this target.
components	The components used to setup Gradle publications.

JVM targets

In addition to common target configuration, jvm targets have a specific function:

Name	Description
withJava()	Includes Java sources into the JVM target's compilations.

Use this function for projects that contain both Java and Kotlin source files. Note that the default source directories for Java sources don't follow the Java plugin's defaults. Instead, they are derived from the Kotlin source sets. For example, if the JVM target has the default name jvm, the paths are src/jvmMain/java (for production Java sources) and src/jvmTest/java for test Java sources. Learn more about Java sources in JVM compilations.

```
kotlin {
    jvm {
        withJava()
    }
}
```

JavaScript targets

The js block describes the configuration of JavaScript targets. It can contain one of two blocks depending on the target execution environment:

Name	Description
browser	Configuration of the browser target.
nodejs	Configuration of the Node.js target.

Learn more about configuring Kotlin/JS projects.

Browser

browser can contain the following configuration blocks:

Name	Description
testRuns	Configuration of test execution.
runTask	Configuration of project running.
webpackTask	Configuration of project bundling with Webpack.
dceTask	Configuration of Dead Code Elimination.
distribution	Path to output files.

```
kotlin {
    js().browser {
        webpackTask { /* ... */ }
        testRuns { /* ... */ }
        dceTask {
            keep("myKotlinJsApplication.org.example.keepFromDce")
        }
        distribution {
            directory = File("$projectDir/customdir/")
        }
    }
}
```

Node.js

nodejs can contain configurations of test and run tasks:

Name	Description
testRuns	Configuration of test execution.
runTask	Configuration of project running.

```
kotlin {
    js().nodejs {
        runTask { /* ... */ }
        testRuns { /* ... */ }
    }
}
```

Native targets

For native targets, the following specific blocks are available:

Name	Description

Name	Description
binaries	Configuration of binaries to produce.
cinterops	Configuration of interop with C libraries.

Binaries

There are the following kinds of binaries:

Name	Description
executable	Product executable.
test	Test executable.
sharedLib	Shared library.
staticLib	Static library.
framework	Objective-C framework.

```
kotlin {
    linuxX64 { // Use your target instead.
        binaries {
            executable {
                // Binary configuration.
            }
        }
    }
}
```

For binary configuration, the following parameters are available:

Name	Description
compilation	The compilation from which the binary is built. By default, test binaries are based on the test compilation while other binaries - on the main compilation.
linkerOpts	Options passed to a system linker during binary building.
baseName	Custom base name for the output file. The final file name will be formed by adding system-dependent prefix and postfix to this base name.
entryPoint	The entry point function for executable binaries. By default, it's main() in the root package.
outputFile	Access to the output file.

Name	Description
linkTask	Access to the link task.
runTask	Access to the run task for executable binaries. For targets other than linuxX64, macosX64, or mingwX64 the value is null.
isStatic	For Objective-C frameworks. Includes a static library instead of a dynamic one.

Kotlin

```
binaries {
    executable("my_executable", listOf(RELEASE)) {
        // Build a binary on the basis of the test compilation.
        compilation = compilations["test"]

        // Custom command line options for the linker.
        linkerOpts = mutableListOf("-L/lib/search/path", "-L/another/search/path", "-lmylib")

        // Base name for the output file.
        baseName = "foo"

        // Custom entry point function.
        entryPoint = "org.example.main"

        // Accessing the output file.
        println("Executable path: ${outputFile.absolutePath}")

        // Accessing the link task.
        linkTask.dependsOn(additionalPreprocessingTask)

        // Accessing the run task.
        // Note that the runTask is null for non-host platforms.
        runTask?.dependsOn(prepareForRun)
    }

    framework("my_framework" listOf(RELEASE)) {
        // Include a static library instead of a dynamic one into the framework.
        isStatic = true
    }
}
```

Groovy

```
binaries {
    executable('my_executable', [RELEASE]) {
        // Build a binary on the basis of the test compilation.
        compilation = compilations.test

        // Custom command line options for the linker.
        linkerOpts = ['-L/lib/search/path', '-L/another/search/path', '-lmylib']

        // Base name for the output file.
        baseName = 'foo'

        // Custom entry point function.
        entryPoint = 'org.example.main'

        // Accessing the output file.
        println("Executable path: ${outputFile.absolutePath}")

        // Accessing the link task.
        linkTask.dependsOn(additionalPreprocessingTask)

        // Accessing the run task.
        // Note that the runTask is null for non-host platforms.
        runTask?.dependsOn(prepareForRun)
    }

    framework('my_framework' [RELEASE]) {
```

```
        // Include a static library instead of a dynamic one into the framework.
        isStatic = true
    }
}
```

Learn more about building native binaries.

CInterops

cinterops is a collection of descriptions for interop with native libraries. To provide an interop with a library, add an entry to cinterops and define its parameters:

Name	Description
defFile	def file describing the native API.
packageName	Package prefix for the generated Kotlin API.
compilerOpts	Options to pass to the compiler by the cinterop tool.
includeDirs	Directories to look for headers.

Learn more how to configure interop with native languages.

Kotlin

```kotlin
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.getByName("main") {
            val myInterop by cinterops.creating {
                // Def-file describing the native API.
                // The default path is src/nativeInterop/cinterop/<interop-name>.def
                defFile(project.file("def-file.def"))

                // Package to place the Kotlin API generated.
                packageName("org.sample")

                // Options to be passed to compiler by cinterop tool.
                compilerOpts("-Ipath/to/headers")

                // Directories for header search (an analogue of the -I<path> compiler option).
                includeDirs.allHeaders("path1", "path2")

                // A shortcut for includeDirs.allHeaders.
                includeDirs("include/directory", "another/directory")
            }

            val anotherInterop by cinterops.creating { /* ... */ }
        }
    }
}
```

Groovy

```groovy
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.main {
            cinterops {
                myInterop {
                    // Def-file describing the native API.
                    // The default path is src/nativeInterop/cinterop/<interop-name>.def
                    defFile project.file("def-file.def")

                    // Package to place the Kotlin API generated.
                    packageName 'org.sample'
```

```
                // Options to be passed to compiler by cinterop tool.
                compilerOpts '-Ipath/to/headers'

                // Directories for header search (an analogue of the -I<path> compiler option).
                includeDirs.allHeaders("path1", "path2")

                // A shortcut for includeDirs.allHeaders.
                includeDirs("include/directory", "another/directory")
            }

            anotherInterop { /* ... */ }
        }
    }
  }
}
```

Android targets

The Kotlin Multiplatform plugin contains two specific functions for android targets. Two functions help you configure build variants:

Name	Description
publishLibraryVariants()	Specifies build variants to publish. Learn more about publishing Android libraries.
publishAllLibraryVariants()	Publishes all build variants.

```
kotlin {
    android {
        publishLibraryVariants("release", "debug")
    }
}
```

Learn more about compilation for Android.

> The android configuration inside kotlin doesn't replace the build configuration of any Android project. Learn more about writing build scripts for Android projects in Android developer documentation.

Source sets

The sourceSets block describes source sets of the project. A source set contains Kotlin source files that participate in compilations together, along with their resources, dependencies, and language settings.

A multiplatform project contains predefined source sets for its targets; developers can also create custom source sets for their needs.

Predefined source sets

Predefined source sets are set up automatically upon creation of a multiplatform project. Available predefined source sets are the following:

Name	Description
commonMain	Code and resources shared between all platforms. Available in all multiplatform projects. Used in all main compilations of a project.
commonTest	Test code and resources shared between all platforms. Available in all multiplatform projects. Used in all test compilations of a project.

Name	Description
<targetName><compilationName>	Target-specific sources for a compilation. <targetName> is the name of a predefined target and <compilationName> is the name of a compilation for this target. Examples: jsTest, jvmMain.

With Kotlin Gradle DSL, the sections of predefined source sets should be marked by getting.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting { /* ... */ }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain { /* ... */ }
    }
}
```

Learn more about source sets.

Custom source sets

Custom source sets are created by the project developers manually. To create a custom source set, add a section with its name inside the sourceSets section. If using Kotlin Gradle DSL, mark custom source sets by creating.

Kotlin

```
kotlin {
    sourceSets {
        val myMain by creating { /* ... */ } // create a new source set by the name 'MyMain'
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        myMain { /* ... */ } // create or configure a source set by the name 'myMain'
    }
}
```

Note that a newly created source set isn't connected to other ones. To use it in the project's compilations, connect it with other source sets.

Source set parameters

Configurations of source sets are stored inside the corresponding blocks of sourceSets. A source set has the following parameters:

Name	Description
kotlin.srcDir	Location of Kotlin source files inside the source set directory.

Name	Description
resources.srcDir	Location of resources inside the source set directory.
dependsOn	Connection with another source set.
dependencies	Dependencies of the source set.
languageSettings	Language settings applied to the source set.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            kotlin.srcDir("src")
            resources.srcDir("res")

            dependencies {
                /* ... */
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            kotlin.srcDir('src')
            resources.srcDir('res')

            dependencies {
                /* ... */
            }
        }
    }
}
```

Compilations

A target can have one or more compilations, for example, for production or testing. There are predefined compilations that are added automatically upon target creation. You can additionally create custom compilations.

To refer to all or some particular compilations of a target, use the compilations object collection. From compilations, you can refer to a compilation by its name.

Learn more about configuring compilations.

Predefined compilations

Predefined compilations are created automatically for each target of a project except for Android targets. Available predefined compilations are the following:

Name	Description
main	Compilation for production sources.

Name	Description
test	Compilation for tests.

Kotlin

```
kotlin {
    jvm {
        val main by compilations.getting {
            output // get the main compilation output
        }

        compilations["test"].runtimeDependencyFiles // get the test runtime classpath
    }
}
```

Groovy

```
kotlin {
    jvm {
        compilations.main.output // get the main compilation output
        compilations.test.runtimeDependencyFiles // get the test runtime classpath
    }
}
```

Custom compilations

In addition to predefined compilations, you can create your own custom compilations. To create a custom compilation, add a new item into the compilations collection. If using Kotlin Gradle DSL, mark custom compilations by creating.

Learn more about creating a custom compilation.

Kotlin

```
kotlin {
    jvm() {
        compilations {
            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
                        /* ... */
                    }
                }

                // Create a test task to run the tests produced by this compilation:
                tasks.register<Test>("integrationTest") {
                    /* ... */
                }
            }
        }
    }
}
```

Groovy

```
kotlin {
    jvm() {
        compilations.create('integrationTest') {
            defaultSourceSet {
                dependencies {
                    /* ... */
                }
            }

            // Create a test task to run the tests produced by this compilation:
            tasks.register('jvmIntegrationTest', Test) {
                /* ... */
```

```
            }
        }
    }
}
```

Compilation parameters

A compilation has the following parameters:

Name	Description
defaultSourceSet	The compilation's default source set.
kotlinSourceSets	Source sets participating in the compilation.
allKotlinSourceSets	Source sets participating in the compilation and their connections via dependsOn().
compilerOptions	Compiler options applied to the compilation. For the list of available options, see Compiler options.
compileKotlinTask	Gradle task for compiling Kotlin sources.
compileKotlinTaskName	Name of compileKotlinTask.
compileAllTaskName	Name of the Gradle task for compiling all sources of a compilation.
output	The compilation output.
compileDependencyFiles	Compile-time dependency files (classpath) of the compilation.
runtimeDependencyFiles	Runtime dependency files (classpath) of the compilation.

Kotlin

```
kotlin {
    jvm {
        val main by compilations.getting {
            compilerOptions.configure {
                // Set up the Kotlin compiler options for the 'main' compilation:
                jvmTarget.set(JvmTarget.JVM_1_8)
            }

            compileKotlinTask // get the Kotlin task 'compileKotlinJvm'
            output // get the main compilation output
        }

        compilations["test"].runtimeDependencyFiles // get the test runtime classpath
    }

    // Configure all compilations of all targets:
    targets.all {
        compilations.all {
            compilerOptions.configure {
                allWarningsAsErrors.set(true)
            }
```

```
        }
    }
}
```

Groovy

```
kotlin {
    jvm {
        compilations.main.compilerOptions.configure {
            // Setup the Kotlin compiler options for the 'main' compilation:
            jvmTarget.set(JvmTarget.JVM_1_8)
        }

        compilations.main.compileKotlinTask // get the Kotlin task 'compileKotlinJvm'
        compilations.main.output // get the main compilation output
        compilations.test.runtimeDependencyFiles // get the test runtime classpath
    }

    // Configure all compilations of all targets:
    targets.all {
        compilations.all {
            compilerOptions.configure {
                allWarningsAsError.set(true)
            }
        }
    }
}
```

Dependencies

The dependencies block of the source set declaration contains the dependencies of this source set.

Learn more about configuring dependencies.

There are four types of dependencies:

Name	Description
api	Dependencies used in the API of the current module.
implementation	Dependencies used in the module but not exposed outside it.
compileOnly	Dependencies used only for compilation of the current module.
runtimeOnly	Dependencies available at runtime but not visible during compilation of any module.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                api("com.example:foo-metadata:1.0")
            }
        }
        val jvm6Main by getting {
            dependencies {
                implementation("com.example:foo-jvm6:1.0")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                api 'com.example:foo-metadata:1.0'
            }
        }
        jvm6Main {
            dependencies {
                implementation 'com.example:foo-jvm6:1.0'
            }
        }
    }
}
```

Additionally, source sets can depend on each other and form a hierarchy. In this case, the dependsOn() relation is used.

Source set dependencies can also be declared in the top-level dependencies block of the build script. In this case, their declarations follow the pattern <sourceSetName><DependencyKind>, for example, commonMainApi.

Kotlin

```
dependencies {
    "commonMainApi"("com.example:foo-common:1.0")
    "jvm6MainApi"("com.example:foo-jvm6:1.0")
}
```

Groovy

```
dependencies {
    commonMainApi 'com.example:foo-common:1.0'
    jvm6MainApi 'com.example:foo-jvm6:1.0'
}
```

Language settings

The languageSettings block of a source set defines certain aspects of project analysis and build. The following language settings are available:

Name	Description
languageVersion	Provides source compatibility with the specified version of Kotlin.
apiVersion	Allows using declarations only from the specified version of Kotlin bundled libraries.
enableLanguageFeature	Enables the specified language feature. The available values correspond to the language features that are currently experimental or have been introduced as such at some point.
optIn	Allows using the specified opt-in annotation.
progressiveMode	Enables the progressive mode.

Kotlin

```
kotlin {
```

```
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.8" // possible values: "1.4", "1.5", "1.6", "1.7", "1.8", "1.9"
            apiVersion = "1.8" // possible values: "1.3", "1.4", "1.5", "1.6", "1.7", "1.8", "1.9"
            enableLanguageFeature("InlineClasses") // language feature name
            optIn("kotlin.ExperimentalUnsignedTypes") // annotation FQ-name
            progressiveMode = true // false by default
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.8' // possible values: '1.4', '1.5', '1.6', '1.7', '1.8', '1.9'
            apiVersion = '1.8' // possible values: '1.3', '1.4', '1.5', '1.6', '1.7', '1.8', '1.9'
            enableLanguageFeature('InlineClasses') // language feature name
            optIn('kotlin.ExperimentalUnsignedTypes') // annotation FQ-name
            progressiveMode = true // false by default
        }
    }
}
```

Kotlin Multiplatform for mobile samples

This is a curated list of cross-platform mobile projects created with Kotlin Multiplatform.

> You can find even more sample projects on GitHub, see the kotlin-multiplatform-mobile topic.
>
> If you want to add your Kotlin Multiplatform project to this topic and help the community, follow the instructions in the GitHub documentation.

Sample name	What's shared?	Popular libraries used	UI Framework	iOS integration	Platform APIs	Tests	Features
Kotlin Multiplatform Mobile Sample	Algorithms	-	XML, SwiftUI	Xcode build phases		-	• expect/actual declarations
KMM RSS Reader	Models, Networking, Data Storage, UI State	SQLDelight, Ktor, DateTime, multiplatform-settings, Napier, kotlinx.serialization	Jetpack Compose, SwiftUI	Xcode build phases		-	• Redux for sharing UI State • Published to Google Play and App Store
kmm-ktor-sample	Networking	Ktor, kotlinx.serialization, Napier	XML, SwiftUI	Xcode build phases	-	-	• Video tutorial

Sample name	What's shared?	Popular libraries used	UI Framework	iOS integration	Platform APIs	Tests	Features
todoapp	Models, Networking, Presentation, Navigation and UI	SQLDelight, Decompose, MVIKotlin, Reaktive	Jetpack Compose, SwiftUI	Xcode build phases	-		• 99% of the code is shared • MVI architectural pattern • Shared UI across Android, Desktop and Web via Compose Multiplatform
mpp-sample-lib	Algorithms	-	-	-		-	• Demonstrates how to create a multiplatform library (tutorial)
KaMPKit	Models, Networking, Data Storage, ViewModels	Koin, SQLDelight, Ktor, DateTime, multiplatform-settings, Kermit	Jetpack Compose, SwiftUI	CocoaPods	-		-
PeopleInSpace	Models, Networking, Data Storage	Koin, SQLDelight, Ktor	Jetpack Compose, SwiftUI	CocoaPods, Swift Packages	-		Target list: • Android Wear OS • iOS • watchOS • macOS Desktop (Compose for Desktop) • Web (Compose for Web) • Web (Kotlin/JS + React Wrapper) • JVM
D-KMP-sample	Networking, Data Storage, ViewModels, Navigation	SQLDelight, Ktor, DateTime, multiplatform-settings	Jetpack Compose, SwiftUI	Xcode build phases	-		• Implements the MVI pattern and the unidirectional data flow • Uses Kotlin's StateFlow to trigger UI layer recompositions

Sample name	What's shared?	Popular libraries used	UI Framework	iOS integration	Platform APIs	Tests	Features
Food2Fork Recipe App	Models, Networking, Data Storage, Interactors	SQLDelight, Ktor, DateTime	Jetpack Compose, SwiftUI	CocoaPods	-	-	-
Bookshelf	Models, Networking, Data Storage	Realm-Kotlin, Ktor, kotlinx.serialization	Jetpack Compose, SwiftUI	CocoaPods	-	-	• Uses Realm for data persistence
Notflix	Models, Networking, Caching, ViewModels	Koin, Ktor, Multiplatform settings, kotlinx.coroutines, kotlinx.serialization, kotlinx.datetime, Napier	Jetpack Compose-Android, Compose Multiplatform-Desktop	-		-	• Modular architecture • Runs on desktop • Sharing viewmodel

Kotlin Multiplatform for mobile FAQ

What is Kotlin Multiplatform for mobile?

Kotlin Multiplatform for mobile is an SDK for cross-platform mobile development. You can develop multiplatform mobile applications and share parts of your applications between Android and iOS, such as core layers, business logic, presentation logic, and more.

Kotlin Mobile uses the multiplatform abilities of Kotlin and the features designed for mobile development, such as CocoaPods integration and the Android Studio Plugin.

You may want to watch this introductory video, in which Kotlin Product Marketing Manager Ekaterina Petrova explains in detail what Kotlin Multiplatform for mobile is and how you can use it in your projects. With Ekaterina, you'll set up an environment and prepare for creating your first cross-platform mobile application with Kotlin Multiplatform.

Can I share UIs with Kotlin Multiplatform?

Yes, you can share UIs using Compose Multiplatform, JetBrains' declarative UI framework based on Kotlin and Jetpack Compose. This framework allows you to create shared UI components for platforms like iOS, Android, desktop, and web, helping you to maintain a consistent user interface across different devices and platforms.

Check out the Compose Multiplatform FAQ to learn more.

What is the Kotlin Multiplatform Mobile plugin?

The Kotlin Multiplatform Mobile plugin for Android Studio helps you develop applications that work on both Android and iOS.

With the Kotlin Multiplatform Mobile plugin, you can:

- Run, test, and debug the iOS part of your application on iOS targets straight from Android Studio.
- Quickly create a new multiplatform project.
- Add a multiplatform module into an existing project.

The Kotlin Multiplatform Mobile plugin works only on macOS. This is because iOS simulators, per the Apple requirement, can run only on macOS but not on any other operating systems, such as Microsoft Windows or Linux.

The good news is that you can work with cross-platform projects on Android even without the Kotlin Multiplatform Mobile plugin. If you are going to work with

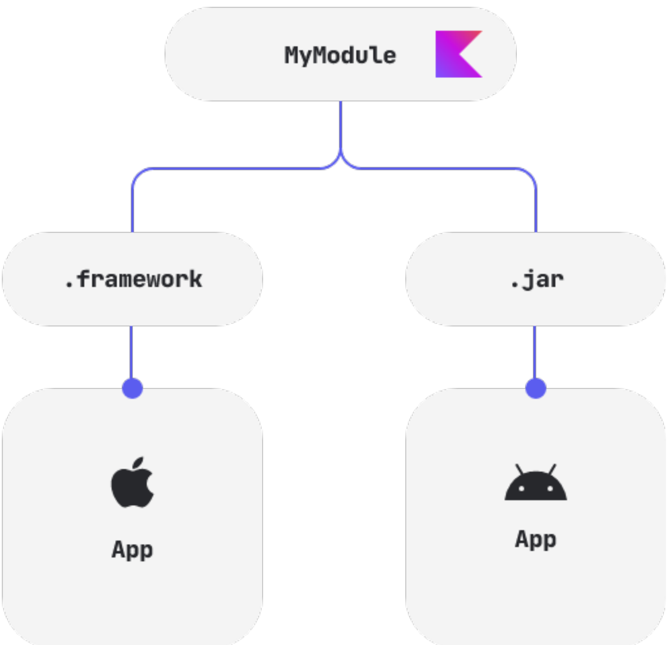
shared code or Android-specific code, you can work on any operating system supported by Android Studio.

What is Kotlin/Native and how does it relate to Kotlin Multiplatform?

Kotlin/Native is a technology for compiling Kotlin code to native binaries, which can run without a virtual machine. It consists of an LLVM-based backend for the Kotlin compiler and a native implementation of the Kotlin standard library.

Kotlin/Native is primarily designed to allow compilation for platforms where virtual machines are not desirable or possible, such as embedded devices and iOS. It is particularly suitable for situations when the developer needs to produce a self-contained program that does not require an additional runtime or virtual machine. And that is exactly the case with iOS development.

Shared code, written in Kotlin, is compiled to JVM bytecode for Android with Kotlin/JVM and to native binaries for iOS with Kotlin/Native. It makes the integration with Kotlin Multiplatform seamless on both platforms.

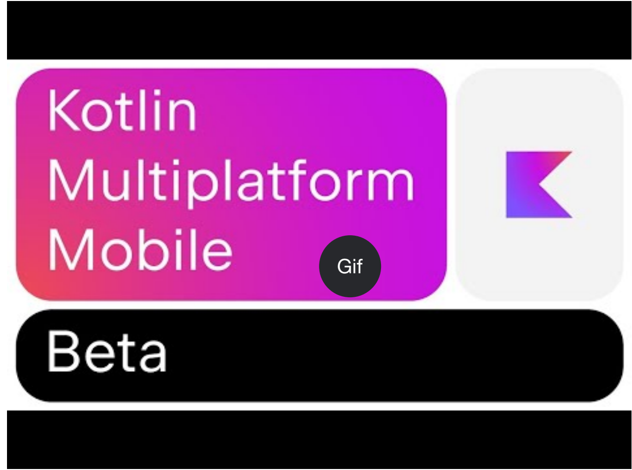


Kotlin/Native and Kotlin/JVM binaries

What are the plans for the technology evolution?

Kotlin Multiplatform is one of the focus areas of the Kotlin roadmap. To see which parts we're working on right now, check out the roadmap details. Most of the recent changes affect the Kotlin Multiplatform and Kotlin/Native sections.

The following video presents the current state and our plans for the Kotlin Multiplatform for mobile development:



Watch video online.

Can I run an iOS application on Microsoft Windows or Linux?

If you want to write iOS-specific code and run an iOS application on a simulated or real device, use a Mac with a macOS (use the Kotlin Multiplatform Mobile plugin for it). This is because iOS simulators can run only on macOS, per the Apple requirement, but cannot run on other operating systems, such as Microsoft Windows or Linux.

If you are going to work with shared code or Android-specific code, you can work on any operating system supported by Android Studio.

Where can I get complete examples to play with?

- Curated samples
- Create a multiplatform app using Ktor and SQLDelight – tutorial

In which IDE should I work on my cross-platform app?

You can work in Android Studio. Android Studio allows the use of the Kotlin Multiplatform Mobile plugin, which is a part of the Kotlin ecosystem. Enable the Kotlin Multiplatform Mobile plugin in Android Studio if you want to write iOS-specific code and launch an iOS application on a simulated or real device. The plugin can be used only on macOS.

Most of our adopters use Android Studio. However, if there is any reason for you not to use it, there is another option: you can use IntelliJ IDEA. IntelliJ IDEA provides the ability to create a multiplatform mobile application from the Project Wizard, but you won't be able to launch an iOS application from the IDE.

How can I write concurrent code in Kotlin Multiplatform projects?

You can easily write concurrent code in your cross-platform mobile projects with the new Kotlin/Native memory manager that lifted previous limitations and aligned the behaviour between Kotlin/JVM and Kotlin/Native. The new memory manager has been enabled by default since Kotlin 1.7.20.

How can I speed up my Kotlin Multiplatform module compilation for iOS?

See these tips for improving Kotlin/Native compilation times.

What platforms do you support?

Kotlin Multiplatform supports development for:

- Android applications and libraries

* Android NDK (ARM64 and ARM32)
* Apple iOS devices and simulators
* Apple watchOS devices and simulators

The Kotlin Multiplatform technology also supports other platforms, including JavaScript, Linux, Windows, and WebAssembly.

Introduce cross-platform mobile development to your team

These recommendations will help you introduce your team to Kotlin Multiplatform for mobile:

* Start with empathy
* Explain how Kotlin Multiplatform works
* Show the value using case studies
* Offer a proof by creating a sample project yourself
* Prepare for questions from your team
* Support your team during the adaptation

Start with empathy

Software development is a team game, with each critical decision needing the approval of all team members. Integrating any cross-platform technology will significantly affect the development process for your mobile application. So before you start integrating Kotlin Multiplatform in your project, you'll need to introduce your team to the technology and guide them gently to see it's worth adopting.

Understanding the people who work on your project is the first step to successful integration. Your boss is responsible for delivering features with the best quality in the shortest time possible. To them, any new technology is a risk. Your colleagues have a different perspective, as well. They have experience building apps with the "native" technology stack. They know how to write the UI and business logic, work with dependencies, test, and debug code in the IDE, and they are already familiar with the language. Switching to a different ecosystem is very uncomfortable, as it always means leaving your comfort zone.

Given all that, be ready to face lots of biases and answer a lot of questions when advocating for the move to Kotlin Multiplatform for mobile. As you do, never lose sight of what your team needs. Some of the advice below might be useful for preparing your pitch.

Explain how it works

At this stage, you need to get rid of any preexisting bad feelings about cross-platform mobile applications and show that using Kotlin Multiplatform in your project is not only possible but also won't bring regular cross-platform problems. You should explain why there won't be any problems, such as:

* Limitations of using all iOS and Android features – Whenever a task cannot be solved in the shared code or whenever you want to use specific native features, you can use the expect/actual pattern to seamlessly write platform-specific code.
* Performance issues – Shared code written in Kotlin is compiled to different output formats for different targets: to Java bytecode for Android and to native binaries for iOS. Thus, there is no additional runtime overhead when it comes to executing this code on platforms, and the performance is comparable to native apps.
* Legacy code problems – No matter how large your project is, your existing code will not prevent you from integrating Kotlin Multiplatform. You can start writing cross-platform code at any moment and connect it to your iOS and Android Apps as a regular dependency, or you can use the code you've already written and simply modify it to be compatible with iOS.

Being able to explain how technology works is important, as nobody likes when a discussion seems to rely on magic. People might think the worst if anything is unclear to them, so be careful not to make the mistake of thinking something is too obvious to warrant explanation. Instead, try to explain all the basic concepts before moving on to the next stage. This document on multiplatform programming could help you systemize your knowledge to prepare for this experience.

Show the value

Understanding how the technology works is necessary, but not enough. Your team needs to see the gains of using it, and the way you present these gains should be related to your product. Kotlin Multiplatform allows you to use a single codebase for the business logic of iOS and Android apps. So if you develop a very thin

client and the majority of the code is UI logic, then the main power of Kotlin Multiplatform Mobile will be unused in your project. However, if your application has complex business logic, for example if you have features like networking, data storage, payments, complex computations, or data synchronization, then this logic could easily be written and shared between iOS and Android so you can experience the real power of the technology.

At this stage, you need to explain the main gains of using Kotlin Multiplatform in your product. One of the ways is to share stories of other companies who already benefit from the technology. The successful experience of these teams, especially ones with similar product objectives, could become a key factor in the final decision.

Citing case studies of different companies who already use Kotlin Multiplatform in production could significantly help you make a compelling argument:

- Chalk.com – The UI for each of the Chalk.com apps is native to the platform, but otherwise almost everything for their apps can be shared with Kotlin Multiplatform for mobile.
- Cash App – A lot of the app's business logic, including the ability to search through all transactions, is implemented with Kotlin Multiplatform for mobile.
- Yandex.Disk – They started out by experimenting with the integration of a small feature, and as the experiment was considered successful, they implemented their whole data synchronization logic in Kotlin Multiplatform for mobile.

Explore the case studies page for inspirational references.

Offer proof

The theory is good, but putting it into practice is ultimately most important. As one option to make your case more convincing, you can take the risky choice of devoting some of your personal free time to creating something with Kotlin Multiplatform and then bringing in the results for your team to discuss. Your prototype could be some sort of test project, which you would write from scratch and which would demonstrate features that are needed in your application. Create a multiplatform app using Ktor and SQLDelight – tutorial can guide you well on this process.

The more relevant examples could be produced by experimenting with your current project. You could take one existing feature implemented in Kotlin and make it cross-platform, or you could even create a new Multiplatform Module in your existing project, take one non-priority feature from the bottom of the backlog, and implement it in the shared module. Make your Android application work on iOS – tutorial provides a step-by-step guide based on a sample project.

The new Kotlin Multiplatform Mobile plugin for Android Studio will allow you to accomplish either of these tasks in the shortest amount of time by using the Kotlin Multiplatform App or Kotlin Multiplatform Library wizards.

Prepare for questions

No matter how detailed your pitch is, your team will have a lot of questions. Listen carefully, and try to answer them all patiently. You might expect the majority of the questions to come from the iOS part of the team, as they are the developers who aren't used to seeing Kotlin in their everyday developer routine. This list of some of the most common questions could help you here:

Q: I heard applications based on cross-platform technologies can be rejected from the AppStore. Is taking this risk worth it?

A: The Apple Store has strict guidelines for application publishing. One of the limitations is that apps may not download, install, or execute code which introduces or changes features or functionality of the app (App Store Review Guideline 2.5.2). This is relevant for some cross-platform technologies, but not for Kotlin Multiplatform. Shared Kotlin code compiles to native binaries with Kotlin/Native, bundles a regular iOS framework into your app, and doesn't provide the ability for dynamic code execution.

Q: Multiplatform projects are built with Gradle, and Gradle has an extremely steep learning curve. Do I need to spend a lot of time now trying to configure my project?

A: There's actually no need. There are various ways to organize the work process around building Kotlin mobile applications. First, only Android developers could be responsible for the builds, in which case the iOS team would only write code or even only consume the resulting artifact. You also can organize some workshops or practice pair programming while facing tasks that require working with Gradle, and this would increase your team's Gradle skills. You can explore different ways of and choose the one that's most appropriate for your team.

Also, in basic scenarios, you simply need to configure your project at the start, and then you just add dependencies to it. The new AS plugin makes configuring your project much easier, so it can now be done in a few clicks.

When only the Android part of the team works with shared code, the iOS developers don't even need to learn Kotlin. But when you are ready for your team to move to the next stage, where everyone contributes to the shared code, making the transition won't take much time. The similarities between the syntax and functionality of Swift and Kotlin greatly reduce the work required to learn how to read and write shared Kotlin code. Try it yourself!

Q: I heard that Kotlin Multiplatform for mobile is experimental technology. Does that mean that we shouldn't use it for production?

A: Experimental status means we and the whole Kotlin community are just trying out an idea, but if it doesn't work, it may be dropped anytime. However, after the

release of Kotlin 1.4, Kotlin Multiplatform for mobile is in Alpha status. This means the Kotlin team is fully committed to working to improve and evolve this technology and will not suddenly drop it. However, before going Beta, there could be some migration issues yet. But even experimental status doesn't prevent a feature from being used successfully in production, as long as you understand all the risks. Check the Kotlin evolution page for information about the stability statuses of Kotlin Multiplatform components.

Q: There are not enough multiplatform libraries to implement the business logic, it's much easier to find native alternatives.

A: Of course, we can't compare the number of multiplatform libraries with React Native, for example. But it took five years for React Native to expand their ecosystem to its current size. Kotlin Multiplatform for mobile is still young, but the ecosystem has tremendous potential as there are already a lot of modern libraries written in Kotlin that can be easily ported to multiplatform.

It's also a great time to be an iOS developer in the Kotlin Multiplatform open-source community because the iOS experience is in demand and there are plenty of opportunities to gain recognition from iOS-specific contributions.

And the more your team digs into the technology, the more interesting and complex their questions will be. Don't worry if you don't have the answers – Kotlin Multiplatform has a large and supportive community in the Kotlin Slack, where a lot of developers who already use it can help you. We would be very thankful if you could share with us the most popular questions asked by your team. This information will help us understand what topics need to be covered in the documentation.

Be supportive

After you decide to use Kotlin Multiplatform, there will be an adaptation period as your team experiments with the technology. And your mission will not be over yet! By providing continuous support for your teammates, you will reduce the time it takes for your team to dive into the technology and achieve their first results.

Here are some tips on how you can support your team at this stage:

- Collect the questions you were asked during the previous stage on the "Kotlin Multiplatform: Frequently asked questions" wiki page and share it with your team.
- Create a #kotlin-multiplatform-support Slack channel and become the most active user there.
- Organize an informal team building event with popcorn and pizza where you watch educational or inspirational videos about Kotlin Multiplatform. "Shipping a Mobile Multiplatform Project on iOS & Android" by Ben Asher & Alec Strong could be a good choice.

The reality is that you probably will not change people's hearts and minds in a day or even a week. But patience and attentiveness to the needs of your colleagues will undoubtedly bring results.

The Kotlin Multiplatform team looks forward to hearing your story.

We'd like to thank the Touchlab team for helping us write this article.

Compatibility guide for Kotlin Multiplatform

This guide summarizes incompatible changes you might encounter while developing projects with Kotlin Multiplatform.

> Mind the deprecation cycle of a specific change in relation to the Kotlin version you have in your projects. The current Stable version of Kotlin is 1.9.0.

New approach to auto-generated targets

What's changed?

Target accessors auto-generated by Gradle are no longer available inside the kotlin.targets block. Use the findByName("targetName") method instead.

Note that such accessors are still available in the kotlin.targets case, for example, kotlin.targets.linuxX64.

What's the best practice now?

Before	Now

Before	Now
```kotlin { targets { configure(['windows', 'linux']) { } } }```	```kotlin { targets { configure([findByName('windows'), findByName('linux')]) { } } }```

When do the changes take effect?

In Kotlin 1.7.20, an error is introduced when using target accessors in the kotlin.targets block.

For more information, see the corresponding issue in YouTrack.

Changes in Gradle input and output compile tasks

What's changed?

Kotlin compile tasks no longer inherit the Gradle AbstractCompile task that has the sourceCompatibility and targetCompatibility inputs, making them unavailable in Kotlin users' scripts.

Other breaking changes in compile tasks:

What's the best practice now?

Before	Now
The SourceTask.stableSources input is no longer available.	Use the sources input instead. Also, the setSource() methods are still available.
The sourceFilesExtensions input was removed.	Compile tasks still implement the PatternFilterable interface. Use its methods for filtering Kotlin sources.
The Gradle destinationDir: File output was deprecated.	Use the destinationDirectory: DirectoryProperty output instead.
The classpath property of the KotlinCompile task is deprecated.	All compile tasks now use the libraries input for a list of libraries required for compilation.

When do the changes take effect?

In Kotlin 1.7.20, inputs are not available, the output is replaced, and the classpath property is deprecated.

For more information, see the corresponding issue in YouTrack.

New configuration names for dependencies on the compilation

What's changed?

Compilation configurations created by the Kotlin Multiplatform Gradle Plugin received new names.

A target in the Kotlin Multiplatform project has two default compilations, main and test. Each of these compilations has its own default source set, for example, jvmMain and jvmTest. Previously the configuration names for the test compilation and its default source set were the same, which might lead to a name clash resulting in issues when a configuration marked with platform-specific attributes is included in another configuration.

Now compilation configurations have an extra Compilation postfix, while projects and plugins that use old hard-coded configuration names no longer compile.

Configuration names for dependencies on the corresponding source set stay the same.

What's the best practice now?

	Before	Now
Dependencies of the jvmMain compilation	`jvm<Scope>`	`jvmCompilation<Scope>`

	`dependencies {`
	`add("jvmImplementation",`
	`"foo.bar.baz:1.2.3")`
	`}`
		`dependencies {`
		`add("jvmCompilationImplementation",`
		`"foo.bar.baz:1.2.3")`
		`}`

Dependencies of the jvmMain source set	`jvmMain<Scope>`	
Dependencies of the jvmTest compilation	`jvmTest<Scope>`	`jvmTestCompilation<Scope>`
Dependencies of the jvmTest source set	`jvmTest<Scope>`	

The available scopes are Api, Implementation, CompileOnly, and RuntimeOnly.

When do the changes take effect?

In Kotlin 1.8.0, an error is introduced when using old configuration names in hard-coded strings.

For more information, see the corresponding issue in YouTrack.

Deprecated Gradle properties for hierarchical structure support

What's changed?

Throughout its evolution, Kotlin was gradually introducing the support for hierarchical structure, in multiplatform projects, an ability to have intermediate source sets between the common source set commonMain and any platform-specific one, for example, jvmMain.

For the transition period, while the toolchain wasn't stable enough, a couple of Gradle properties were introduced, allowing granular opt-ins and opt-outs.

Since Kotlin 1.6.20, the hierarchical project structure support has been enabled by default. However, these properties were kept for opting out in case of blocking issues. After processing all the feedback, we're now starting to phase out those properties completely.

The following properties are now deprecated and will be removed in Kotlin 1.9.20:

- kotlin.internal.mpp.hierarchicalStructureByDefault
- kotlin.mpp.enableCompatibilityMetadataVariant
- kotlin.mpp.hierarchicalStructureSupport
- kotlin.mpp.enableGranularSourceSetsMetadata
- kotlin.native.enableDependencyPropagation

What's the best practice now?

* Remove these properties from your gradle.properties and local.properties files.
* Avoid setting them programmatically in the Gradle build scripts or your Gradle plugins.
* In case deprecated properties are set by some third-party Gradle plugin used in your build, ask the plugin maintainers not to set these properties.

As the default behavior of the Kotlin toolchain doesn't include such properties since 1.6.20, we don't expect any serious impact from removing them. Most possible consequences will be visible immediately after the project rebuild.

If you're a library author and want to be extra safe, check that consumers can work with your library.

When do the changes take effect?

In 1.8.20, the Kotlin Gradle plugin shows a warning if the build sets these properties. Starting with Kotlin 1.9.0, the properties are silently ignored.

In the unlikely case you face some problems after removing these properties, create an issue in YouTrack.

Deprecated support of multiplatform libraries published in the legacy mode

What's changed?

Previously, we have deprecated the legacy mode in Kotlin Multiplatform projects preventing the publication of "legacy" binaries and encouraged you to migrate your projects to the hierarchical structure.

To continue phasing out "legacy" binaries from the ecosystem, starting with Kotlin 1.9.0, the use of legacy libraries is also discouraged. If your project uses dependencies on legacy libraries, you'll see the following warning:

```
The dependency group:artifact:1.0 was published in the legacy mode. Support for such dependencies will be removed in the future
```

What's the best practice now?

If you use multiplatform libraries, most of them have already migrated to the "hierarchical structure" mode, so you only need to update the library version. See the documentation of the respective libraries for details.

If the library doesn't support non-legacy binaries yet, you can contact the maintainers and tell them about this compatibility issue.

If you're a library author, update the Kotlin Gradle plugin to the latest version and ensure you've fixed the deprecated Gradle properties.

The Kotlin team is eager to help the ecosystem migrate, so if you face any issues, don't hesitate to create an issue in YouTrack.

When do the changes take effect?

* 1.9.0: introduce a deprecation warning for dependencies on legacy libraries
* 1.9.20: raise the warning for dependencies on legacy libraries to an error
* > 1.9.20: the support for dependencies on legacy libraries is removed. Using such dependencies can cause build failures

Deprecated API for adding Kotlin source sets directly to the Kotlin compilation

What's changed?

The access to KotlinCompilation.source has been deprecated. A code like this produces a deprecation warning:

```
kotlin {
    jvm()
    js()
    ios()

    sourceSets {
        val commonMain by getting
        val myCustomIntermediateSourceSet by creating {
            dependsOn(commonMain)
        }

        targets["jvm"].compilations["main"].source(myCustomIntermediateSourceSet)
    }
```

```
}
```

What's the best practice now?

To replace KotlinCompilation.source(someSourceSet), add the dependsOn relation from the default source set of the KotlinCompilation to someSourceSet. We recommend referring to the source directly using by getting, which is shorter and more readable. However, you can also use KotlinCompilation.defaultSourceSet.dependsOn(someSourceSet), which is applicable in all cases.

You can change the code above in one of the following ways:

```
kotlin {
    jvm()
    js()
    ios()

    sourceSets {
        val commonMain by getting
        val myCustomIntermediateSourceSet by creating {
            dependsOn(commonMain)
        }

        // Option #1. Shorter and more readable, use it when possible:
       val jvmMain by getting { // Usually, the name of the default source set
                                // is a simple concatenation of the target name and the compilation name
            dependsOn(myCustomIntermediateSourceSet)
        }

        // Option #2. Generic solution, use it if your build script requires a more advanced approach:
        targets["jvm"].compilations["main"].defaultSourceSet.dependsOn(myCustomIntermediateSourceSet)
    }
}
```

When do the changes take effect?

In 1.9.0, the use of KotlinComplation.source produces a deprecation warning. This API will be removed in Kotlin 1.9.20 and later, leading to "unresolved reference" errors on the KotlinCompilation.source calls.

Migration from kotlin-js Gradle plugin to kotlin-multiplatform Gradle plugin

What's changed?

Starting with Kotlin 1.9.0, the kotlin-js Gradle plugin is deprecated. Basically, it duplicated the functionality of the kotlin-multiplatform plugin with the js() target and shared the same implementation under the hood. Such overlap created confusion and increased maintenance load on the Kotlin team. We encourage you to migrate to the kotlin-multiplatform Gradle plugin with the js() target instead.

What's the best practice now?

1. Remove the kotlin-js Gradle plugin from your project and apply kotlin-multiplatform in the settings.gradle.kts file if you're using the pluginManagement block:

kotlin-js

```
// settings.gradle.kts
pluginManagement {
    plugins {
        // Remove the following line:
        kotlin("js") version "1.9.0"
    }

    repositories {
        // ...
    }
}
```

kotlin-multiplatform

```
// settings.gradle.kts
pluginManagement {
    plugins {
        // Add the following line instead:
```

```
        kotlin("multiplatform") version "1.9.0"
    }

    repositories {
        // ...
    }
}
```

In case you're using a different way of applying plugins, see the Gradle documentation for migration instructions.

2. Move your source files from the main and test folders to the jsMain and jsTest folders in the same directory.

3. Adjust dependency declarations:

   - We recommend using the sourceSets block and configuring dependencies of respective source sets, jsMain for production dependencies and jsTest for test dependencies. See Adding dependencies for more details.

   - However, if you want to declare your dependencies in a top-level block, change declarations from api("group:artifact:1.0") to add("jsMainApi", "group:artifact:1.0") and so on.

     > In this case, make sure that the top-level dependencies block comes after the kotlin block. Otherwise, you'll get an error "Configuration not found".

You can change the code in your build.gradle.kts file in one of the following ways:

kotlin-js

```
// build.gradle.kts
plugins {
    kotlin("js") version "1.9.0"
}

dependencies {
    testImplementation(kotlin("test"))
    implementation("org.jetbrains.kotlinx:kotlinx-html:0.8.0")
}

kotlin {
    js {
        // ...
    }
}
```

kotlin-multiplatform

```
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "1.9.0"
}

kotlin {
    js {
        // ...
    }

    // Option #1. Declare dependencies in the `sourceSets` block:
    sourceSets {
        val jsMain by getting {
            dependencies {
                // No need for the `js` prefix here, you can just copy and paste it from the top-level block
                implementation("org.jetbrains.kotlinx:kotlinx-html:0.8.0")
            }
        }
    }
}

dependencies {
   // Option #2. Add the `js` prefix to the dependency declaration:
   add("jsTestImplementation", kotlin("test"))
}
```

4. The DSL provided by the Kotlin Gradle plugin inside the kotlin block remains unchanged in most cases. However, if you were referring to low-level Gradle entities, like tasks and configurations, by names, you now need to adjust them, usually by adding the js prefix. For example, you can find the browserTest task under the name jsBrowserTest.

When do the changes take effect?

In 1.9.0, the use of the kotlin-js Gradle plugin produces a deprecation warning.

Rename of android target to androidTarget

What's changed?

We continue our efforts to stabilize Kotlin Multiplatform. An essential step in this way is to provide first-class support for the Android target. In the future, this support will be provided via a separate plugin, developed by the Android team from Google.

To open the way for the new solution from Google, we're renaming the android block to androidTarget in the current Kotlin DSL in 1.9.0. This is a temporary change that is necessary to free the short android name for the upcoming DSL from Google.

What's the best practice now?

Rename all the occurrences of the android block to androidTarget. When the new plugin for the Android target support is available, migrate to the DSL from Google. It will be the preferred option to work with Android in Kotlin Multiplatform projects.

When do the changes take effect?

In Kotlin 1.9.0, a deprecation warning is introduced when the android name is used in Kotlin Multiplatform projects.

Kotlin Multiplatform Mobile plugin releases

Since Kotlin Multiplatform Mobile is now in Beta, we are working on stabilizing the corresponding plugin for Android Studio and will be regularly releasing new versions that include new features, improvements, and bug fixes.

Ensure that you have the latest version of the Kotlin Multiplatform Mobile plugin!

Update to the new release

Android Studio will suggest updating to a new Kotlin Multiplatform Mobile plugin release as soon as it is available. If you accept the suggestion, it will automatically update the plugin to the latest version. You'll need to restart Android Studio to complete the plugin installation.

You can check the plugin version and update it manually in Settings/Preferences | Plugins.

You need a compatible version of Kotlin for the plugin to work correctly. You can find compatible versions in the release details. You can check your Kotlin version and update it in Settings/Preferences | Plugins or in Tools | Kotlin | Configure Kotlin Plugin Updates.

> If you do not have a compatible version of Kotlin installed, the Kotlin Multiplatform Mobile plugin will be disabled. You will need to update your Kotlin version, and then enable the plugin in Settings/Preferences | Plugins.

Release details

The following table lists the details of the latest Kotlin Multiplatform Mobile plugin releases:

Release info	Release highlights	Compatible Kotlin version

Release info	Release highlights	Compatible Kotlin version
0.6.0 Released: 24 May, 2023	• Support of the new Canary Android Studio Hedgehog. • Updated versions of Kotlin, Gradle, and libraries in the Multiplatform project. • Applied new targetHierarchy.default() in the Multiplatform project. • Applied source set name suffixes to platform-specific files in the Multiplatform project.	• Any of Kotlin plugin versions
0.5.3 Released: 12 April, 2023	• Updated Kotlin and Compose versions. • Fixed an Xcode project scheme parsing. • Added a scheme product type check. • iosApp scheme is now selected by default if presented.	• Any of Kotlin plugin versions
0.5.2 Released: 30 January, 2023	• Fixed a problem with Kotlin/Native debugger (slow Spotlight indexing). • Fixed Kotlin/Native debugger in multimodule projects. • New build for Android Studio Giraffe 2022.3.1 Canary. • Added provisioning flags for an iOS app build. • Added inherited paths to the Framework Search Paths option in a generated iOS project.	• Any of Kotlin plugin versions
0.5.1 Released: 30 November, 2022	• Fixed new project generation: delete an excess "app" directory.	• Kotlin 1.7.0—*
0.5.0 Released: 22 November, 2022	• Changed the default option for iOS framework distribution: now it is Regular framework. • Moved MyApplicationTheme to a separate file in a generated Android project. • Updated generated Android project. • Fixed an issue with unexpected erasing of new project directory.	• Kotlin 1.7.0—*

Release info	Release highlights	Compatible Kotlin version
0.3.4 Released: 12 September, 2022	• Migrated Android app to Jetpack Compose. • Removed outdated HMPP flags. • Removed package name from Android manifest. • Updated .gitignore for Xcode projects. • Updated wizard project for better illustration expect/actual. • Updated compatibility with Canary build of Android Studio. • Updated minimum Android SDK to 21 for Android app. • Fixed an issue with the first launch after installation Xcode. • Fixed an issues with Apple run configuration on M1. • Fixed an issue with local.properties on Windows OS. • Fixed an issue with Kotlin/Native debugger on Canary build of Android Studio.	• Kotlin 1.7.0—1.7.*
0.3.3 Released: 9 June, 2022	• Updated dependency on Kotlin IDE plugin 1.7.0.	• Kotlin 1.7.0—1.7.*
0.3.2 Released: 4 April, 2022	• Fixed the performance problem with the iOS application debug on Android Studio 2021.2 and 2021.3.	• Kotlin 1.5.0—1.6.*
0.3.1 Released: 15 February, 2022	• Enabled M1 iOS simulator in Kotlin Multiplatform Mobile wizards. • Improved performance for indexing XcProjects: KT-49777, KT-50779. • Build scripts clean up: use kotlin("test") instead of kotlin("test-common") and kotlin("test-annotations-common"). • Increase compatibility range with Kotlin plugin version. • Fixed the problem with JVM debug on Windows host. • Fixed the problem with the invalid version after disabling the plugin.	• Kotlin 1.5.0—1.6.*

Release info	Release highlights	Compatible Kotlin version
0.3.0 Released: 16 November, 2021	• New Kotlin Multiplatform Library wizard. • Support for the new type of Kotlin Multiplatform library distribution: XCFramework. • Enabled hierarchical project structure for new cross-platform mobile projects. • Support for explicit iOS targets declaration. • Enabled Kotlin Multiplatform Mobile plugin wizards on non-Mac machines. • Support for subfolders in the Kotlin Multiplatform module wizard. • Support for Xcode Assets.xcassets file. • Fixed the plugin classloader exception. • Updated the CocoaPods Gradle Plugin template. • Kotlin/Native debugger type evaluation improvements. • Fixed iOS device launching with Xcode 13.	• Kotlin 1.6.0
0.2.7 Released: August 2, 2021	• Added Xcode configuration option for AppleRunConfiguration. • Added support Apple M1 simulators. • Added information about Xcode integration options in Project Wizard. • Added error notification after a project with CocoaPods was generated, but the CocoaPods gem has not been installed. • Added support Apple M1 simulator target in generated shared module with Kotlin 1.5.30 • Cleared generated Xcode project with Kotlin 1.5.20. • Fixed launching Xcode Release configuration on a real iOS device. • Fixed simulator launching with Xcode 12.5.	• Kotlin 1.5.10
0.2.6 Released: June 10, 2021	• Compatibility with Android Studio Bumblebee Canary 1. • Support for Kotlin 1.5.20: using the new framework-packing task for Kotlin/Native in the Project Wizard.	• Kotlin 1.5.10
0.2.5 Released: May 25, 2021	• Fixed compatibility with Android Studio Arctic Fox 2020.3.1 Beta 1 and higher	• Kotlin 1.5.10
0.2.4 Released: May 5, 2021	Use this version of the plugin with Android Studio 4.2 or Android Studio 2020.3.1 Canary 8 or higher. • Compatibility with Kotlin 1.5.0. • Ability to use the CocoaPods dependency manager in the Kotlin Multiplatform module for iOS integration.	• Kotlin 1.5.0

Release info	Release highlights	Compatible Kotlin version
0.2.3 Released: April 5, 2021	• The Project Wizard: improvements in naming modules. • Ability to use the CocoaPods dependency manager in the Project Wizard for iOS integration. • Better readability of gradle.properties in new projects. • Sample tests are no longer generated if "Add sample tests for Shared Module" is unchecked. • Fixes and other improvements.	• Kotlin 1.4.30
0.2.2 Released: March 3, 2021	• Ability to open Xcode-related files in Xcode. • Ability to set up a location for the Xcode project file in the iOS run configuration. • Support for Android Studio 2020.3.1 Canary 8. • Fixes and other improvements.	• Kotlin 1.4.30
0.2.1 Released: February 15, 2021	Use this version of the plugin with Android Studio 4.2. • Infrastructure improvements. • Fixes and other improvements.	• Kotlin 1.4.30
0.2.0 Released: November 23, 2020	• Support for iPad devices. • Support for custom scheme names that are configured in Xcode. • Ability to add custom build steps for the iOS run configuration. • Ability to debug a custom Kotlin/Native binary. • Simplified the code generated by Kotlin Multiplatform Mobile Wizards. • Removed support for the Kotlin Android Extensions plugin, which is deprecated in Kotlin 1.4.20. • Fixed saving physical device configuration after disconnecting from the host. • Other fixes and improvements.	• Kotlin 1.4.20
0.1.3 Released: October 2, 2020	• Added compatibility with iOS 14 and Xcode 12. • Fixed naming in platform tests created by the Kotlin Multiplatform Mobile Wizard.	• Kotlin 1.4.10 • Kotlin 1.4.20
0.1.2 Released: September 29, 2020	• Fixed compatibility with Kotlin 1.4.20-M1. • Enabled error reporting to JetBrains by default.	• Kotlin 1.4.10 • Kotlin 1.4.20

Release info	Release highlights	Compatible Kotlin version
0.1.1 Released: September 10, 2020	• Fixed compatibility with Android Studio Canary 8 and higher.	• Kotlin 1.4.10 • Kotlin 1.4.20
0.1.0 Released: August 31, 2020	• The first version of the Kotlin Multiplatform Mobile plugin. Learn more in the blog post.	• Kotlin 1.4.0 • Kotlin 1.4.10

Get started with Kotlin/JVM

This tutorial demonstrates how to use IntelliJ IDEA for creating a console application.

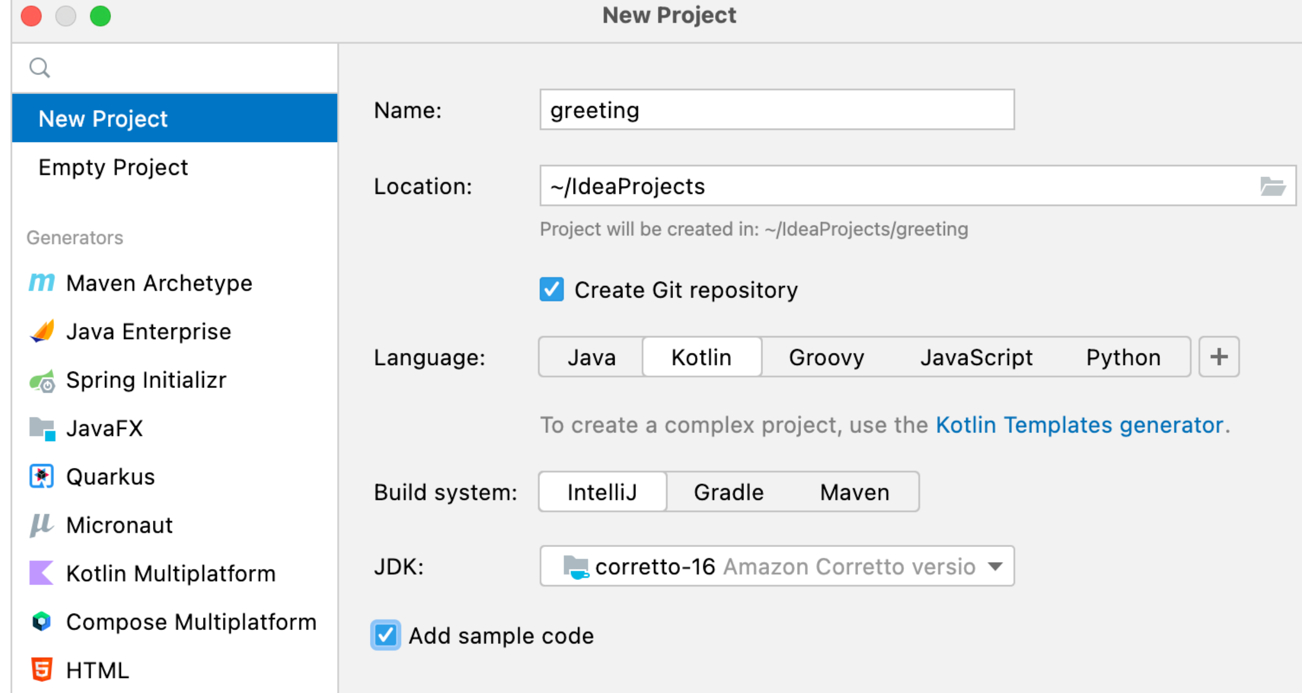
To get started, first download and install the latest version of IntelliJ IDEA.

Create a project

1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select New Project.
3. Name the new project and change its location if necessary.

> Select the Create Git repository checkbox to place the new project under version control. You will be able to do it later at any time.

4. From the Language list, select Kotlin.



Create a console application

5. Select the IntelliJ build system. It's a native builder that doesn't require downloading additional artifacts.

   If you want to create a more complex project that needs further configuration, select Maven or Gradle. For Gradle, choose a language for the build script: Kotlin or Groovy.

6. From the JDK list, select the JDK that you want to use in your project.

   - If the JDK is installed on your computer, but not defined in the IDE, select Add JDK and specify the path to the JDK home directory.

   - If you don't have the necessary JDK on your computer, select Download JDK.

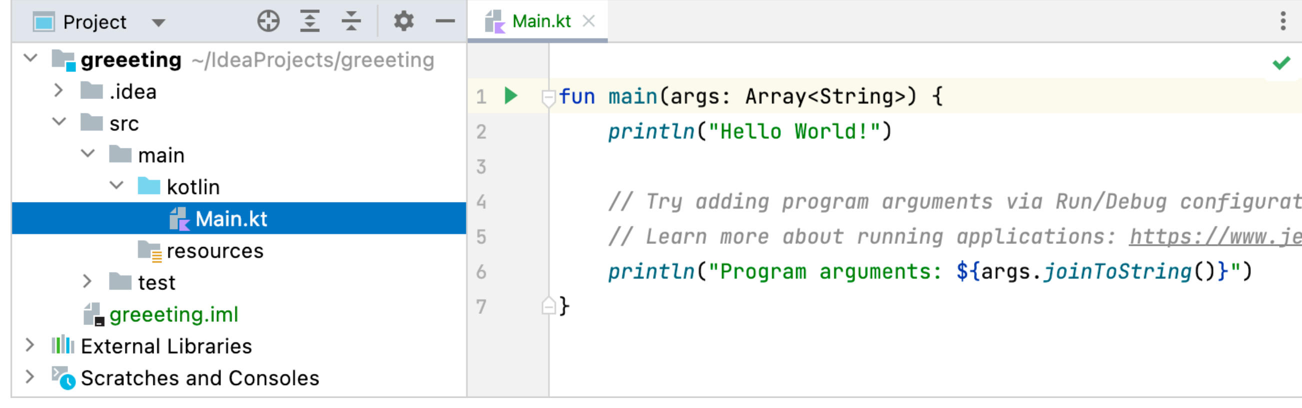
7. Enable the Add sample code option to create a file with a sample "Hello World!" application.

8. Click Create.

> If you chose the Gradle build system, you have in your project a build script file: build.gradle(.kts). It includes the kotlin("jvm") plugin and dependencies required for your console application. Make sure that you use the latest version of the plugin:
>
> ```
> plugins {
>     kotlin("jvm") version "1.9.0"
>     application
> }
> ```

Create an application

1. Open the Main.kt file in src/main/kotlin.
   The src directory contains Kotlin source files and resources. The Main.kt file contains sample code that will print Hello World!.



Main.kt with main fun

2. Modify the code so that it requests your name and says Hello to you alone, and not to the whole world:

   - Introduce a local variable name with the keyword val. It will get its value from an input where you will enter your name – readln().

     > The readln() function is available since Kotlin 1.6.0.
     > Ensure that you have installed the latest version of the Kotlin plugin.

   - Use a string template by adding a dollar sign $ before this variable name directly in the text output like this – $name.

```
fun main() {
    println("What's your name?")
    val name = readln()
    println("Hello, $name!")
```

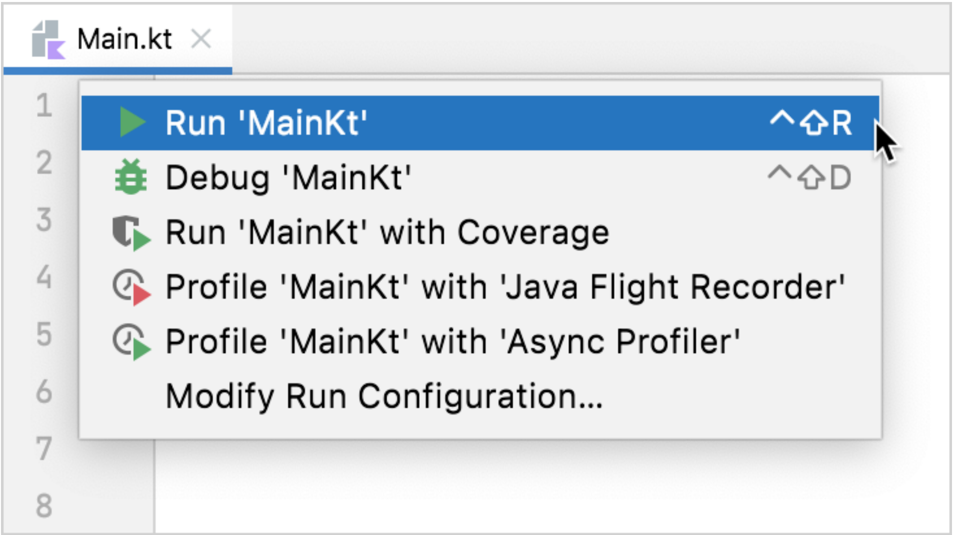
```
}
```

```
fun main() {
    println("What's your name?")
    val name = readln()
    println("Hello, $name!")
}
```

Updated main fun

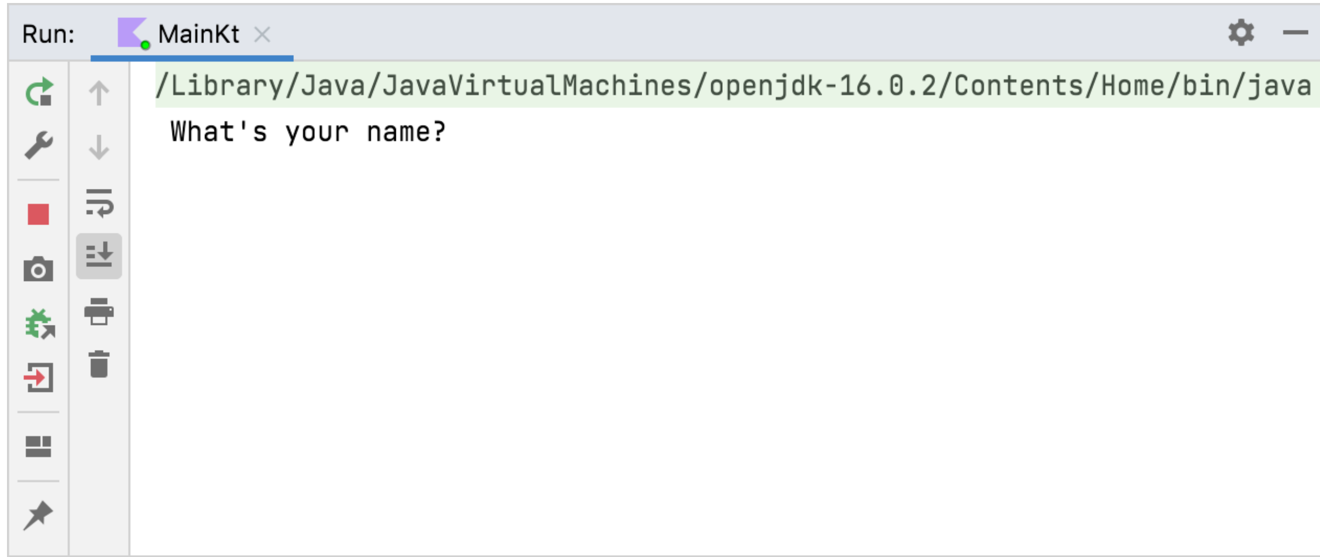
Run the application

Now the application is ready to run. The easiest way to do this is to click the green Run icon in the gutter and select Run 'MainKt'.



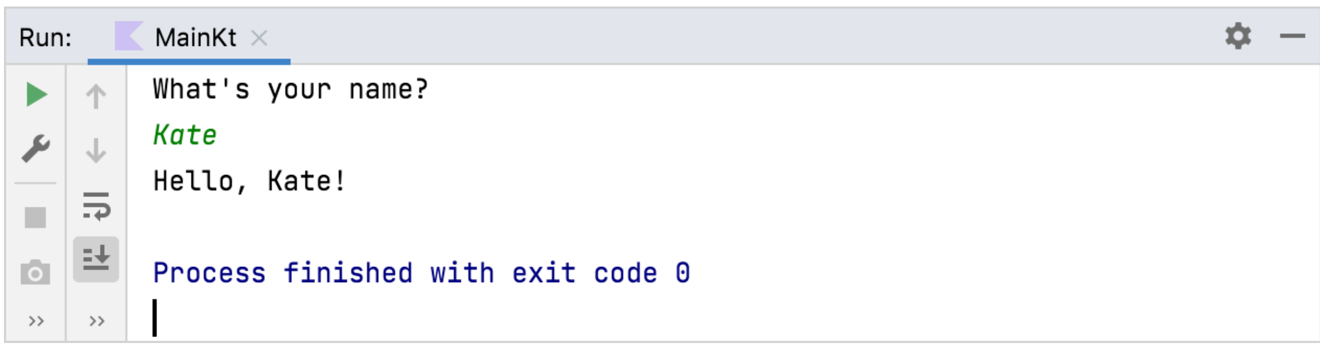
Running a console app

You can see the result in the Run tool window.



Kotlin run output

Enter your name and accept the greetings from your application!



Kotlin run output

Congratulations! You have just run your first Kotlin application.

What's next?

Once you've created this application, you can start to dive deeper into Kotlin syntax:

- Add sample code from Kotlin examples
- Install the JetBrains Academy plugin for IDEA and complete exercises from the Kotlin Koans course

Comparison to Java

Some Java issues addressed in Kotlin

Kotlin fixes a series of issues that Java suffers from:

- Null references are controlled by the type system.
- No raw types
- Arrays in Kotlin are invariant

* Kotlin has proper function types, as opposed to Java's SAM-conversions
* Use-site variance without wildcards
* Kotlin does not have checked exceptions

What Java has that Kotlin does not

* Checked exceptions
* Primitive types that are not classes. The byte-code uses primitives where possible, but they are not explicitly available.
* Static members are replaced with companion objects, top-level functions, extension functions, or @JvmStatic.
* Wildcard-types are replaced with declaration-site variance and type projections.
* Ternary-operator a ? b : c is replaced with if expression.

What Kotlin has that Java does not

* Lambda expressions + Inline functions = performant custom control structures
* Extension functions
* Null-safety
* Smart casts
* String templates
* Properties
* Primary constructors
* First-class delegation
* Type inference for variable and property types
* Singletons
* Declaration-site variance & Type projections
* Range expressions
* Operator overloading
* Companion objects
* Data classes
* Separate interfaces for read-only and mutable collections
* Coroutines

What's next?

Learn how to:

* Perform typical tasks with strings in Java and Kotlin.
* Perform typical tasks with collections in Java and Kotlin.
* Handle nullability in Java and Kotlin.

Calling Java from Kotlin

Kotlin is designed with Java interoperability in mind. Existing Java code can be called from Kotlin in a natural way, and Kotlin code can be used from Java rather smoothly as well. In this section, we describe some details about calling Java code from Kotlin.

Pretty much all Java code can be used without any issues:

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }
    // Operator conventions work as well:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

Getters and setters

Methods that follow the Java conventions for getters and setters (no-argument methods with names starting with get and single-argument methods with names starting with set) are represented as properties in Kotlin. Such properties are also called synthetic properties. Boolean accessor methods (where the name of the getter starts with is and the name of the setter starts with set) are represented as properties which have the same name as the getter method.

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY // call setFirstDayOfWeek()
    }
    if (!calendar.isLenient) { // call isLenient()
        calendar.isLenient = true // call setLenient()
    }
}
```

calendar.firstDayOfWeek above is an example of a synthetic property.

Note that, if the Java class only has a setter, it isn't visible as a property in Kotlin because Kotlin doesn't support set-only properties.

Java synthetic property references

> This feature is Experimental. It may be dropped or changed at any time. We recommend that you use it only for evaluation purposes.

Starting from Kotlin 1.8.20, you can create references to Java synthetic properties. Consider the following Java code:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Kotlin has always allowed you to write person.age, where age is a synthetic property. Now, you can also create references to Person::age and person::age. The

same applies for name, as well.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12), Person("Peter", 11))
    Persons
         // Call a reference to Java synthetic property:
        .sortedBy(Person::age)
         // Call Java getter via the Kotlin property syntax:
        .forEach { person -> println(person.name) }
}
```

How to enable Java synthetic property references

To enable this feature, set the -language-version 1.9 compiler option. In a Gradle project, you can do so by adding the following to your build.gradle(.kts):

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion
            = org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
}
```

Methods returning void

If a Java method returns void, it will return Unit when called from Kotlin. If by any chance someone uses that return value, it will be assigned at the call site by the Kotlin compiler since the value itself is known in advance (being Unit).

Escaping for Java identifiers that are keywords in Kotlin

Some of the Kotlin keywords are valid identifiers in Java: in, object, is, and other. If a Java library uses a Kotlin keyword for a method, you can still call the method escaping it with the backtick (`) character:

```
foo.`is`(bar)
```

Null-safety and platform types

Any reference in Java may be null, which makes Kotlin's requirements of strict null-safety impractical for objects coming from Java. Types of Java declarations are treated in Kotlin in a specific manner and called platform types. Null-checks are relaxed for such types, so that safety guarantees for them are the same as in Java (see more below).

Consider the following examples:

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)
```

When you call methods on variables of platform types, Kotlin does not issue nullability errors at compile time, but the call may fail at runtime, because of a null-

pointer exception or an assertion that Kotlin generates to prevent nulls from propagating:

```
item.substring(1) // allowed, throws an exception if item == null
```

Platform types are non-denotable, meaning that you can't write them down explicitly in the language. When a platform value is assigned to a Kotlin variable, you can rely on the type inference (the variable will have an inferred platform type then, as item has in the example above), or you can choose the type you expect (both nullable and non-null types are allowed):

```
val nullable: String? = item // allowed, always works
val notNull: String = item // allowed, may fail at runtime
```

If you choose a non-null type, the compiler will emit an assertion upon assignment. This prevents Kotlin's non-null variables from holding nulls. Assertions are also emitted when you pass platform values to Kotlin functions expecting non-null values and in other cases. Overall, the compiler does its best to prevent nulls from propagating far through the program although sometimes this is impossible to eliminate entirely, because of generics.

Notation for platform types

As mentioned above, platform types can't be mentioned explicitly in the program, so there's no syntax for them in the language. Nevertheless, the compiler and IDE need to display them sometimes (for example, in error messages or parameter info), so there is a mnemonic notation for them:

- T! means "T or T?",
- (Mutable)Collection<T>! means "Java collection of T may be mutable or not, may be nullable or not",
- Array<(out) T>! means "Java array of T (or a subtype of T), nullable or not"

Nullability annotations

Java types that have nullability annotations are represented not as platform types, but as actual nullable or non-null Kotlin types. The compiler supports several flavors of nullability annotations, including:

- JetBrains (@Nullable and @NotNull from the org.jetbrains.annotations package)
- JSpecify (org.jspecify.nullness)
- Android (com.android.annotations and android.support.annotations)
- JSR-305 (javax.annotation, more details below)
- FindBugs (edu.umd.cs.findbugs.annotations)
- Eclipse (org.eclipse.jdt.annotation)
- Lombok (lombok.NonNull)
- RxJava 3 (io.reactivex.rxjava3.annotations)

You can specify whether the compiler reports a nullability mismatch based on the information from specific types of nullability annotations. Use the compiler option -Xnullability-annotations=@<package-name>:<report-level>. In the argument, specify the fully qualified nullability annotations package and one of these report levels:

- ignore to ignore nullability mismatches
- warn to report warnings
- strict to report errors.

See the full list of supported nullability annotations in the Kotlin compiler source code.

Annotating type arguments and type parameters

You can annotate the type arguments and type parameters of generic types to provide nullability information for them as well.

> All examples in the section use JetBrains nullability annotations from the org.jetbrains.annotations package.

Type arguments

Consider these annotations on a Java declaration:

```
@NotNull
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) { ... }
```

They result in the following signature in Kotlin:

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { ... }
```

When the @NotNull annotation is missing from a type argument, you get a platform type instead:

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { ... }
```

Kotlin also takes into account nullability annotations on type arguments of base classes and interfaces. For example, there are two Java classes with the signatures provided below:

```
public class Base<T> {}
```

```
public class Derived extends Base<@Nullable String> {}
```

In the Kotlin code, passing the instance of Derived where the Base<String> is assumed produces the warning.

```
fun takeBaseOfNotNullStrings(x: Base<String>) {}

fun main() {
    takeBaseOfNotNullStrings(Derived()) // warning: nullability mismatch
}
```

The upper bound of Derived is set to Base<String?>, which is different from Base<String>.

Learn more about Java generics in Kotlin.

Type parameters

By default, the nullability of plain type parameters in both Kotlin and Java is undefined. In Java, you can specify it using nullability annotations. Let's annotate the type parameter of the Base class:

```
public class Base<@NotNull T> {}
```

When inheriting from Base, Kotlin expects a non-nullable type argument or type parameter. Thus, the following Kotlin code produces a warning:

```
class Derived<K> : Base<K> {} // warning: K has undefined nullability
```

You can fix it by specifying the upper bound K : Any.

Kotlin also supports nullability annotations on the bounds of Java type parameters. Let's add bounds to Base:

```
public class BaseWithBound<T extends @NotNull Number> {}
```

Kotlin translates this just as follows:

```
class BaseWithBound<T : Number> {}
```

So passing nullable type as a type argument or type parameter produces a warning.

Annotating type arguments and type parameters works with the Java 8 target or higher. The feature requires that the nullability annotations support the TYPE_USE target (org.jetbrains.annotations supports this in version 15 and above). Pass the -Xtype-enhancement-improvements-strict-mode compiler option to report errors in Kotlin code that uses nullability which deviates from the nullability annotations from Java.

Note: If a nullability annotation supports other targets that are applicable to a type in addition to the TYPE_USE target, then TYPE_USE takes priority. For example, if @Nullable has both TYPE_USE and METHOD targets, the Java method signature @Nullable String[] f() becomes fun f(): Array<String?>! in Kotlin.

JSR-305 support

The @Nonnull annotation defined in JSR-305 is supported for denoting nullability of Java types.

If the @Nonnull(when = ...) value is When.ALWAYS, the annotated type is treated as non-null; When.MAYBE and When.NEVER denote a nullable type; and When.UNKNOWN forces the type to be platform one.

A library can be compiled against the JSR-305 annotations, but there's no need to make the annotations artifact (e.g. jsr305.jar) a compile dependency for the library consumers. The Kotlin compiler can read the JSR-305 annotations from a library without the annotations present on the classpath.

Custom nullability qualifiers (KEEP-79) are also supported (see below).

Type qualifier nicknames

If an annotation type is annotated with both @TypeQualifierNickname and JSR-305 @Nonnull (or its another nickname, such as @CheckForNull), then the annotation type is itself used for retrieving precise nullability and has the same meaning as that nullability annotation:

```
@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
@CheckForNull // a nickname to another type qualifier nickname
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@MyNonnull String x);
    // in Kotlin (strict mode): `fun foo(x: String): String?`

    String bar(List<@MyNonnull String> x);
    // in Kotlin (strict mode): `fun bar(x: List<String>!): String!`
}
```

Type qualifier defaults

@TypeQualifierDefault allows introducing annotations that, when being applied, define the default nullability within the scope of the annotated element.

Such annotation type should itself be annotated with both @Nonnull (or its nickname) and @TypeQualifierDefault(...) with one or more ElementType values:

- ElementType.METHOD for return types of methods
- ElementType.PARAMETER for value parameters
- ElementType.FIELD for fields
- ElementType.TYPE_USE for any type including type arguments, upper bounds of type parameters and wildcard types

The default nullability is used when a type itself is not annotated by a nullability annotation, and the default is determined by the innermost enclosing element annotated with a type qualifier default annotation with the ElementType matching the type usage.

```
@Nonnull
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NonNullApi {
}

@Nonnull(when = When.MAYBE)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE_USE})
public @interface NullableApi {
}

@NullableApi
interface A {
    String foo(String x); // fun foo(x: String?): String?
```

```
    @NotNullApi // overriding default from the interface
    String bar(String x, @Nullable String y); // fun bar(x: String, y: String?): String

    // The List<String> type argument is seen as nullable because of `@NullableApi`
    // having the `TYPE_USE` element type:
    String baz(List<String> x); // fun baz(List<String?>?): String?

    // The type of `x` parameter remains platform because there's an explicit
    // UNKNOWN-marked nullability annotation:
    String qux(@Nonnull(when = When.UNKNOWN) String x); // fun baz(x: String!): String?
}
```

The types in this example only take place with the strict mode enabled; otherwise, the platform types remain. See the @UnderMigration annotation and Compiler configuration sections.

Package-level default nullability is also supported:

```
// FILE: test/package-info.java
@NonNullApi // declaring all types in package 'test' as non-nullable by default
package test;
```

@UnderMigration annotation

The @UnderMigration annotation (provided in a separate artifact kotlin-annotations-jvm) can be used by library maintainers to define the migration status for the nullability type qualifiers.

The status value in @UnderMigration(status = ...) specifies how the compiler treats inappropriate usages of the annotated types in Kotlin (e.g. using a @MyNullable-annotated type value as non-null):

- MigrationStatus.STRICT makes annotation work as any plain nullability annotation, i.e. report errors for the inappropriate usages and affect the types in the annotated declarations as they are seen in Kotlin
- MigrationStatus.WARN: the inappropriate usages are reported as compilation warnings instead of errors, but the types in the annotated declarations remain platform
- MigrationStatus.IGNORE makes the compiler ignore the nullability annotation completely

A library maintainer can add @UnderMigration status to both type qualifier nicknames and type qualifier defaults:

```
@Nonnull(when = When.ALWAYS)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
@UnderMigration(status = MigrationStatus.WARN)
public @interface NonNullApi {
}

// The types in the class are non-null, but only warnings are reported
// because `@NonNullApi` is annotated `@UnderMigration(status = MigrationStatus.WARN)`
@NonNullApi
public class Test {}
```

The migration status of a nullability annotation is not inherited by its type qualifier nicknames but is applied to its usages in default type qualifiers.

If a default type qualifier uses a type qualifier nickname and they are both @UnderMigration, the status from the default type qualifier is used.

Compiler configuration

The JSR-305 checks can be configured by adding the -Xjsr305 compiler flag with the following options (and their combination):

- -Xjsr305={strict|warn|ignore} to set up the behavior for non-@UnderMigration annotations. Custom nullability qualifiers, especially @TypeQualifierDefault, are already spread among many well-known libraries, and users may need to migrate smoothly when updating to the Kotlin version containing JSR-305 support. Since Kotlin 1.1.60, this flag only affects non-@UnderMigration annotations.
- -Xjsr305=under-migration:{strict|warn|ignore} to override the behavior for the @UnderMigration annotations. Users may have different view on the migration status for the libraries: they may want to have errors while the official migration status is WARN, or vice versa, they may wish to postpone errors reporting for

some until they complete their migration.

- -Xjsr305=@<fq.name>:{strict|warn|ignore} to override the behavior for a single annotation, where <fq.name> is the fully qualified class name of the annotation. May appear several times for different annotations. This is useful for managing the migration state for a particular library.

The strict, warn and ignore values have the same meaning as those of MigrationStatus, and only the strict mode affects the types in the annotated declarations as they are seen in Kotlin.

> Note: the built-in JSR-305 annotations @Nonnull, @Nullable and @CheckForNull are always enabled and affect the types of the annotated declarations in Kotlin, regardless of compiler configuration with the -Xjsr305 flag.

For example, adding -Xjsr305=ignore -Xjsr305=under-migration:ignore -Xjsr305=@org.library.MyNullable:warn to the compiler arguments makes the compiler generate warnings for inappropriate usages of types annotated by @org.library.MyNullable and ignore all other JSR-305 annotations.

The default behavior is the same to -Xjsr305=warn. The strict value should be considered experimental (more checks may be added to it in the future).

Mapped types

Kotlin treats some Java types specifically. Such types are not loaded from Java "as is", but are mapped to corresponding Kotlin types. The mapping only matters at compile time, the runtime representation remains unchanged. Java's primitive types are mapped to corresponding Kotlin types (keeping platform types in mind):

Java type	Kotlin type
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

Some non-primitive built-in classes are also mapped:

Java type	Kotlin type
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!

Java type	Kotlin type
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.annotation.Annotation	kotlin.Annotation!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Java's boxed primitive types are mapped to nullable Kotlin types:

Java type	Kotlin type
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Character	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

Note that a boxed primitive type used as a type parameter is mapped to a platform type: for example, List<java.lang.Integer> becomes a List<Int!> in Kotlin.

Collection types may be read-only or mutable in Kotlin, so Java's collections are mapped as follows (all Kotlin types in this table reside in the package kotlin.collections):

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable)Iterator<T>!
Iterable<T>	Iterable<T>	MutableIterable<T>	(Mutable)Iterable<T>!
Collection<T>	Collection<T>	MutableCollection<T>	(Mutable)Collection<T>!
Set<T>	Set<T>	MutableSet<T>	(Mutable)Set<T>!
List<T>	List<T>	MutableList<T>	(Mutable)List<T>!
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)ListIterator<T>!
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map<K, V>!
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K,V>	(Mutable)Map.(Mutable)Entry<K, V>!

Java's arrays are mapped as mentioned below:

Java type	Kotlin type
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

> The static members of these Java types are not directly accessible on the companion objects of the Kotlin types. To call them, use the full qualified names of the Java types, e.g. java.lang.Integer.toHexString(foo).

Java generics in Kotlin

Kotlin's generics are a little different from Java's (see Generics). When importing Java types to Kotlin, the following conversions are done:

- Java's wildcards are converted into type projections:
  - Foo<? extends Bar> becomes Foo<out Bar!>!
  - Foo<? super Bar> becomes Foo<in Bar!>!
- Java's raw types are converted into star projections:
  - List becomes List<*>! that is List<out Any?>!

Like Java's, Kotlin's generics are not retained at runtime: objects do not carry information about actual type arguments passed to their constructors. For example, ArrayList<Integer>() is indistinguishable from ArrayList<Character>(). This makes it impossible to perform is-checks that take generics into account. Kotlin only allows is-checks for star-projected generic types:

```
if (a is List<Int>) // Error: cannot check if it is really a List of Ints
```

```
// but
if (a is List<*>) // OK: no guarantees about the contents of the list
```

Java arrays

Arrays in Kotlin are invariant, unlike Java. This means that Kotlin won't let you assign an Array<String> to an Array<Any>, which prevents a possible runtime failure. Passing an array of a subclass as an array of superclass to a Kotlin method is also prohibited, but for Java methods this is allowed through platform types of the form Array<(out) String>!.

Arrays are used with primitive datatypes on the Java platform to avoid the cost of boxing/unboxing operations. As Kotlin hides those implementation details, a workaround is required to interface with Java code. There are specialized classes for every type of primitive array (IntArray, DoubleArray, CharArray, and so on) to handle this case. They are not related to the Array class and are compiled down to Java's primitive arrays for maximum performance.

Suppose there is a Java method that accepts an int array of indices:

```
public class JavaArrayExample {
    public void removeIndices(int[] indices) {
        // code here...
    }
}
```

To pass an array of primitive values, you can do the following in Kotlin:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // passes int[] to method
```

When compiling to the JVM bytecode, the compiler optimizes access to arrays so that there's no overhead introduced:

```
val array = arrayOf(1, 2, 3, 4)
array[1] = array[1] * 2 // no actual calls to get() and set() generated
for (x in array) { // no iterator created
    print(x)
}
```

Even when you navigate with an index, it does not introduce any overhead:

```
for (i in array.indices) { // no iterator created
    array[i] += 2
}
```

Finally, in-checks have no overhead either:

```
if (i in array.indices) { // same as (i >= 0 && i < array.size)
    print(array[i])
}
```

Java varargs

Java classes sometimes use a method declaration for the indices with a variable number of arguments (varargs):

```
public class JavaArrayExample {

    public void removeIndicesVarArg(int... indices) {
        // code here...
    }
}
```

In that case you need to use the spread operator * to pass the IntArray:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

Operators

Since Java has no way of marking methods for which it makes sense to use the operator syntax, Kotlin allows using any Java methods with the right name and signature as operator overloads and other conventions (invoke() etc.) Calling Java methods using the infix call syntax is not allowed.

Checked exceptions

In Kotlin, all exceptions are unchecked, meaning that the compiler does not force you to catch any of them. So, when you call a Java method that declares a checked exception, Kotlin does not force you to do anything:

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // Java would require us to catch IOException here
    }
}
```

Object methods

When Java types are imported into Kotlin, all the references of the type java.lang.Object are turned into Any. Since Any is not platform-specific, it only declares toString(), hashCode() and equals() as its members, so to make other members of java.lang.Object available, Kotlin uses extension functions.

wait()/notify()

Methods wait() and notify() are not available on references of type Any. Their usage is generally discouraged in favor of java.util.concurrent. If you really need to call these methods, you can cast to java.lang.Object:

```
(foo as java.lang.Object).wait()
```

getClass()

To retrieve the Java class of an object, use the java extension property on a class reference:

```
val fooClass = foo::class.java
```

The code above uses a bound class reference. You can also use the javaClass extension property:

```
val fooClass = foo.javaClass
```

clone()

To override clone(), your class needs to extend kotlin.Cloneable:

```
class Example : Cloneable {
    override fun clone(): Any { ... }
}
```

Don't forget about Effective Java, 3rd Edition, Item 13: Override clone judiciously.

finalize()

To override finalize(), all you need to do is simply declare it, without using the override keyword:

```
class C {
    protected fun finalize() {
        // finalization logic
    }
```

```
}
```

According to Java's rules, finalize() must not be private.

Inheritance from Java classes

At most one Java class (and as many Java interfaces as you like) can be a supertype for a class in Kotlin.

Accessing static members

Static members of Java classes form "companion objects" for these classes. You can't pass such a "companion object" around as a value but can access the members explicitly, for example:

```
if (Character.isLetter(a)) { ... }
```

To access static members of a Java type that is mapped to a Kotlin type, use the full qualified name of the Java type: java.lang.Integer.bitCount(foo).

Java reflection

Java reflection works on Kotlin classes and vice versa. As mentioned above, you can use instance::class.java, ClassName::class.java or instance.javaClass to enter Java reflection through java.lang.Class. Do not use ClassName.javaClass for this purpose because it refers to ClassName's companion object class, which is the same as ClassName.Companion::class.java and not ClassName::class.java.

For each primitive type, there are two different Java classes, and Kotlin provides ways to get both. For example, Int::class.java will return the class instance representing the primitive type itself, corresponding to Integer.TYPE in Java. To get the class of the corresponding wrapper type, use Int::class.javaObjectType, which is equivalent of Java's Integer.class.

Other supported cases include acquiring a Java getter/setter method or a backing field for a Kotlin property, a KProperty for a Java field, a Java method or constructor for a KFunction and vice versa.

SAM conversions

Kotlin supports SAM conversions for both Java and Kotlin interfaces. This support for Java means that Kotlin function literals can be automatically converted into implementations of Java interfaces with a single non-default method, as long as the parameter types of the interface method match the parameter types of the Kotlin function.

You can use this for creating instances of SAM interfaces:

```
val runnable = Runnable { println("This runs in a runnable") }
```

...and in method calls:

```
val executor = ThreadPoolExecutor()
// Java signature: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

If the Java class has multiple methods taking functional interfaces, you can choose the one you need to call by using an adapter function that converts a lambda to a specific SAM type. Those adapter functions are also generated by the compiler when needed:

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

SAM conversions only work for interfaces, not for abstract classes, even if those also have just a single abstract method.

Using JNI with Kotlin

To declare a function that is implemented in native (C or C++) code, you need to mark it with the external modifier:

```
external fun foo(x: Int): Double
```

The rest of the procedure works in exactly the same way as in Java.

You can also mark property getters and setters as external:

```
var myProperty: String
    external get
    external set
```

Behind the scenes, this will create two functions getMyProperty and setMyProperty, both marked as external.

Using Lombok-generated declarations in Kotlin

You can use Java's Lombok-generated declarations in Kotlin code. If you need to generate and use these declarations in the same mixed Java/Kotlin module, you can learn how to do this on the Lombok compiler plugin's page. If you call such declarations from another module, then you don't need to use this plugin to compile that module.

Calling Kotlin from Java

Kotlin code can be easily called from Java. For example, instances of a Kotlin class can be seamlessly created and operated in Java methods. However, there are certain differences between Java and Kotlin that require attention when integrating Kotlin code into Java. On this page, we'll describe the ways to tailor the interop of your Kotlin code with its Java clients.

Properties

A Kotlin property is compiled to the following Java elements:

- a getter method, with the name calculated by prepending the get prefix
- a setter method, with the name calculated by prepending the set prefix (only for var properties)
- a private field, with the same name as the property name (only for properties with backing fields)

For example, var firstName: String compiles to the following Java declarations:

```
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

If the name of the property starts with is, a different name mapping rule is used: the name of the getter will be the same as the property name, and the name of the setter will be obtained by replacing is with set. For example, for a property isOpen, the getter will be called isOpen() and the setter will be called setOpen(). This rule applies for properties of any type, not just Boolean.

Package-level functions

All the functions and properties declared in a file app.kt inside a package org.example, including extension functions, are compiled into static methods of a Java class named org.example.AppKt.

```
// app.kt
package org.example
```

```
class Util

fun getTime() { /*...*/ }
```

```
// Java
new org.example.Util();
org.example.AppKt.getTime();
```

To set a custom name to the generated Java class, use the @JvmName annotation:

```
@file:JvmName("DemoUtils")

package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java
new org.example.Util();
org.example.DemoUtils.getTime();
```

Having multiple files with the same generated Java class name (the same package and the same name or the same @JvmName annotation) is normally an error. However, the compiler can generate a single Java facade class which has the specified name and contains all the declarations from all the files which have that name. To enable the generation of such a facade, use the @JvmMultifileClass annotation in all such files.

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getTime() { /*...*/ }
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getDate() { /*...*/ }
```

```
// Java
org.example.Utils.getTime();
org.example.Utils.getDate();
```

Instance fields

If you need to expose a Kotlin property as a field in Java, annotate it with the @JvmField annotation. The field will have the same visibility as the underlying property. You can annotate a property with @JvmField if it:

- has a backing field
- is not private
- does not have open, override or const modifiers
- is not a delegated property

```
class User(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(User user) {
        return user.ID;
    }
}
```

Late-Initialized properties are also exposed as fields. The visibility of the field will be the same as the visibility of lateinit property setter.

Static fields

Kotlin properties declared in a named object or a companion object will have static backing fields either in that named object or in the class containing the companion object.

Usually these fields are private but they can be exposed in one of the following ways:

- @JvmField annotation
- lateinit modifier
- const modifier

Annotating such a property with @JvmField makes it a static field with the same visibility as the property itself.

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// Java
Key.COMPARATOR.compare(key1, key2);
// public static final field in Key class
```

A late-initialized property in an object or a companion object has a static backing field with the same visibility as the property setter.

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java
Singleton.provider = new Provider();
// public static non-final field in Singleton class
```

Properties declared as const (in classes as well as at the top level) are turned into static fields in Java:

```
// file example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

In Java:

```
int constant = Obj.CONST;
```

```
int max = ExampleKt.MAX;
int version = C.VERSION;
```

Static methods

As mentioned above, Kotlin represents package-level functions as static methods. Kotlin can also generate static methods for functions defined in named objects or companion objects if you annotate those functions as @JvmStatic. If you use this annotation, the compiler will generate both a static method in the enclosing class of the object and an instance method in the object itself. For example:

```
class C {
    companion object {
        @JvmStatic fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

Now, callStatic() is static in Java while callNonStatic() is not:

```

C.callStatic(); // works fine
C.callNonStatic(); // error: not a static method
C.Companion.callStatic(); // instance method remains
C.Companion.callNonStatic(); // the only way it works
```

Same for named objects:

```
object Obj {
    @JvmStatic fun callStatic() {}
    fun callNonStatic() {}
}
```

In Java:

```

Obj.callStatic(); // works fine
Obj.callNonStatic(); // error
Obj.INSTANCE.callNonStatic(); // works, a call through the singleton instance
Obj.INSTANCE.callStatic(); // works too
```

Starting from Kotlin 1.3, @JvmStatic applies to functions defined in companion objects of interfaces as well. Such functions compile to static methods in interfaces. Note that static method in interfaces were introduced in Java 1.8, so be sure to use the corresponding targets.

```
interface ChatBot {
    companion object {
        @JvmStatic fun greet(username: String) {
            println("Hello, $username")
        }
    }
}
```

@JvmStatic annotation can also be applied on a property of an object or a companion object making its getter and setter methods static members in that object or the class containing the companion object.

Default methods in interfaces

> Default methods are available only for targets JVM 1.8 and above.

Starting from JDK 1.8, interfaces in Java can contain default methods. To make all non-abstract members of Kotlin interfaces default for the Java classes implementing them, compile the Kotlin code with the -Xjvm-default=all compiler option.

Here is an example of a Kotlin interface with a default method:

```
// compile with -Xjvm-default=all

interface Robot {
    fun move() { println("~walking~") }  // will be default in the Java interface
    fun speak(): Unit
}
```

The default implementation is available for Java classes implementing the interface.

```
//Java implementation
public class C3PO implements Robot {
    // move() implementation from Robot is available implicitly
    @Override
    public void speak() {
        System.out.println("I beg your pardon, sir");
    }
}
```

```
C3PO c3po = new C3PO();
c3po.move(); // default implementation from the Robot interface
c3po.speak();
```

Implementations of the interface can override default methods.

```
//Java
public class BB8 implements Robot {
    //own implementation of the default method
    @Override
    public void move() {
        System.out.println("~rolling~");
    }

    @Override
    public void speak() {
        System.out.println("Beep-beep");
    }
}
```

> Prior to Kotlin 1.4, to generate default methods, you could use the @JvmDefault annotation on these methods. Compiling with -Xjvm-default=all in 1.4+ generally works as if you annotated all non-abstract methods of interfaces with @JvmDefaultand compiled with -Xjvm-default=enable. However, there are cases when their behavior differs. Detailed information about the changes in default methods generation in Kotlin 1.4 is provided in this post on the Kotlin blog.

Compatibility modes for default methods

If there are clients that use your Kotlin interfaces compiled without the -Xjvm-default=all option, then they may be binary-incompatible with the code compiled with this option. To avoid breaking the compatibility with such clients, use the -Xjvm-default=all mode and mark interfaces with the @JvmDefaultWithCompatibility annotation. This allows you to add this annotation to all interfaces in the public API once, and you won't need to use any annotations for new non-public code.

> Starting from Kotlin 1.6.20, you can compile modules in the default mode (the -Xjvm-default=disable compiler option) against modules compiled with the -Xjvm-default=all or -Xjvm-default=all-compatibility modes.

Learn more about compatibility modes:

disable

Default behavior. Do not generate JVM default methods and prohibit @JvmDefault annotation usage.

all

Generate JVM default methods for all interface declarations with bodies in the module. Do not generate DefaultImpls stubs for interface declarations with bodies, which are generated by default in the disable mode.

If interface inherits a method with body from an interface compiled in the disable mode and doesn't override it, then a DefaultImpls stub will be generated for it.

Breaks binary compatibility if some client code relies on the presence of DefaultImpls classes.

> If interface delegation is used, all interface methods are delegated. The only exception are methods annotated with the deprecated @JvmDefault annotation.

all-compatibility

In addition to the all mode, generate compatibility stubs in the DefaultImpls classes. Compatibility stubs could be useful for library and runtime authors to keep backward binary compatibility for existing clients compiled against previous library versions. all and all-compatibility modes are changing the library ABI surface that clients will use after the recompilation of the library. In that sense, clients might be incompatible with previous library versions. This usually means that you need a proper library versioning, for example, major version increase in SemVer.

The compiler generates all the members of DefaultImpls with the @Deprecated annotation: you shouldn't use these members in Java code, because the compiler generates them only for compatibility purposes.

In case of inheritance from a Kotlin interface compiled in all or all-compatibility modes, DefaultImpls compatibility stubs will invoke the default method of the interface with standard JVM runtime resolution semantics.

Perform additional compatibility checks for classes inheriting generic interfaces where in some cases additional implicit method with specialized signatures was generated in the disable mode: unlike in the disable mode, the compiler will report an error if you don't override such method explicitly and don't annotate the class with @JvmDefaultWithoutCompatibility (see this YouTrack issue for more details).

Visibility

The Kotlin visibility modifiers map to Java in the following way:

- private members are compiled to private members
- private top-level declarations are compiled to package-local declarations
- protected remains protected (note that Java allows accessing protected members from other classes in the same package and Kotlin doesn't, so Java classes will have broader access to the code)
- internal declarations become public in Java. Members of internal classes go through name mangling, to make it harder to accidentally use them from Java and to allow overloading for members with the same signature that don't see each other according to Kotlin rules
- public remains public

KClass

Sometimes you need to call a Kotlin method with a parameter of type KClass. There is no automatic conversion from Class to KClass, so you have to do it manually by invoking the equivalent of the Class<T>.kotlin extension property:

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

Handling signature clashes with @JvmName

Sometimes we have a named function in Kotlin, for which we need a different JVM name in the bytecode. The most prominent example happens due to type erasure:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

These two functions can not be defined side-by-side, because their JVM signatures are the same: filterValid(Ljava/util/List;)Ljava/util/List;. If we really want them to have the same name in Kotlin, we can annotate one (or both) of them with @JvmName and specify a different name as an argument:

```
fun List<String>.filterValid(): List<String>
```

```
@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

From Kotlin they will be accessible by the same name filterValid, but from Java it will be filterValid and filterValidInt.

The same trick applies when we need to have a property x alongside with a function getX():

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

To change the names of generated accessor methods for properties without explicitly implemented getters and setters, you can use @get:JvmName and @set:JvmName:

```
@get:JvmName("x")
@set:JvmName("changeX")
var x: Int = 23
```

Overloads generation

Normally, if you write a Kotlin function with default parameter values, it will be visible in Java only as a full signature, with all parameters present. If you wish to expose multiple overloads to Java callers, you can use the @JvmOverloads annotation.

The annotation also works for constructors, static methods, and so on. It can't be used on abstract methods, including methods defined in interfaces.

```
class Circle @JvmOverloads constructor(centerX: Int, centerY: Int, radius: Double = 1.0) {
    @JvmOverloads fun draw(label: String, lineWidth: Int = 1, color: String = "red") { /*...*/ }
}
```

For every parameter with a default value, this will generate one additional overload, which has this parameter and all parameters to the right of it in the parameter list removed. In this example, the following will be generated:

```
// Constructors:
Circle(int centerX, int centerY, double radius)
Circle(int centerX, int centerY)

// Methods
void draw(String label, int lineWidth, String color) { }
void draw(String label, int lineWidth) { }
void draw(String label) { }
```

Note that, as described in Secondary constructors, if a class has default values for all constructor parameters, a public constructor with no arguments will be generated for it. This works even if the @JvmOverloads annotation is not specified.

Checked exceptions

Kotlin does not have checked exceptions. So, normally the Java signatures of Kotlin functions do not declare exceptions thrown. Thus, if you have a function in Kotlin like this:

```
// example.kt
package demo

fun writeToFile() {
    /*...*/
    throw IOException()
}
```

And you want to call it from Java and catch the exception:

```
// Java
try {
```

```
    demo.Example.writeToFile();
} catch (IOException e) {
    // error: writeToFile() does not declare IOException in the throws list
    // ...
}
```

You get an error message from the Java compiler, because writeToFile() does not declare IOException. To work around this problem, use the @Throws annotation in Kotlin:

```
@Throws(IOException::class)
fun writeToFile() {
    /*...*/
    throw IOException()
}
```

Null-safety

When calling Kotlin functions from Java, nobody prevents us from passing null as a non-null parameter. That's why Kotlin generates runtime checks for all public functions that expect non-nulls. This way we get a NullPointerException in the Java code immediately.

Variant generics

When Kotlin classes make use of declaration-site variance, there are two options of how their usages are seen from the Java code. For example, imagine you have the following class and two functions that use it:

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

A naive way of translating these functions into Java would be this:

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

The problem is that in Kotlin you can write unboxBase(boxDerived(Derived())) but in Java that would be impossible because in Java the class Box is invariant in its parameter T, and thus Box<Derived> is not a subtype of Box<Base>. To make this work in Java, you would have to define unboxBase as follows:

```
Base unboxBase(Box<? extends Base> box) { ... }
```

This declaration uses Java's wildcards types (? extends Base) to emulate declaration-site variance through use-site variance, because it is all Java has.

To make Kotlin APIs work in Java, the compiler generates Box<Super> as Box<? extends Super> for covariantly defined Box (or Foo<? super Bar> for contravariantly defined Foo) when it appears as a parameter. When it's a return value, wildcards are not generated, because otherwise Java clients will have to deal with them (and it's against the common Java coding style). Therefore, the functions from our example are actually translated as follows:

```
// return type - no wildcards
Box<Derived> boxDerived(Derived value) { ... }

// parameter - wildcards
Base unboxBase(Box<? extends Base> box) { ... }
```

When the argument type is final, there's usually no point in generating the wildcard, so Box<String> is always Box<String>, no matter what position it takes.

If you need wildcards where they are not generated by default, use the @JvmWildcard annotation:

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// is translated to
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

In the opposite case, if you don't need wildcards where they are generated, use @JvmSuppressWildcards:

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// is translated to
// Base unboxBase(Box<Base> box) { ... }
```

> @JvmSuppressWildcards can be used not only on individual type arguments, but on entire declarations, such as functions or classes, causing all wildcards inside them to be suppressed.

Translation of type Nothing

The type Nothing is special, because it has no natural counterpart in Java. Indeed, every Java reference type, including java.lang.Void, accepts null as a value, and Nothing doesn't accept even that. So, this type cannot be accurately represented in the Java world. This is why Kotlin generates a raw type where an argument of type Nothing is used:

```
fun emptyList(): List<Nothing> = listOf()
// is translated to
// List emptyList() { ... }
```

Get started with Spring Boot and Kotlin

Get started with Spring Boot and Kotlin by completing this tutorial: it walks you through the process of creating a simple application with Spring Boot and adding a database to store the information.

Going through these four steps, you'll learn a lot of essential features of the Kotlin language:

1. Create a Spring Boot project
2. Add a data class to Spring Boot project
3. Add database support for the Spring Boot project
4. Use Spring Data CrudRepository for database access

Next step

Start by creating a Spring Boot project with Kotlin using IntelliJ IDEA.

See also

Look through our Java to Kotlin (J2K) interop and migration guides:

- Calling Java from Kotlin and Calling Kotlin from Java
- Collections in Java and Kotlin
- Strings in Java and Kotlin

Join the community

- Kotlin slack: get an invitation and join the #spring and #server channels
- Stack Overflow: subscribe to the "kotlin", "spring-kotlin", or "ktor" tags
- Kotlin YouTube channel: subscribe and watch videos about Kotlin with Spring

Create a Spring Boot project with Kotlin

The first part of the tutorial shows you how to create a Spring Boot project in IntelliJ IDEA using Project Wizard.

Before you start

Download and install the latest version of IntelliJ IDEA Ultimate Edition.

Create a Spring Boot project

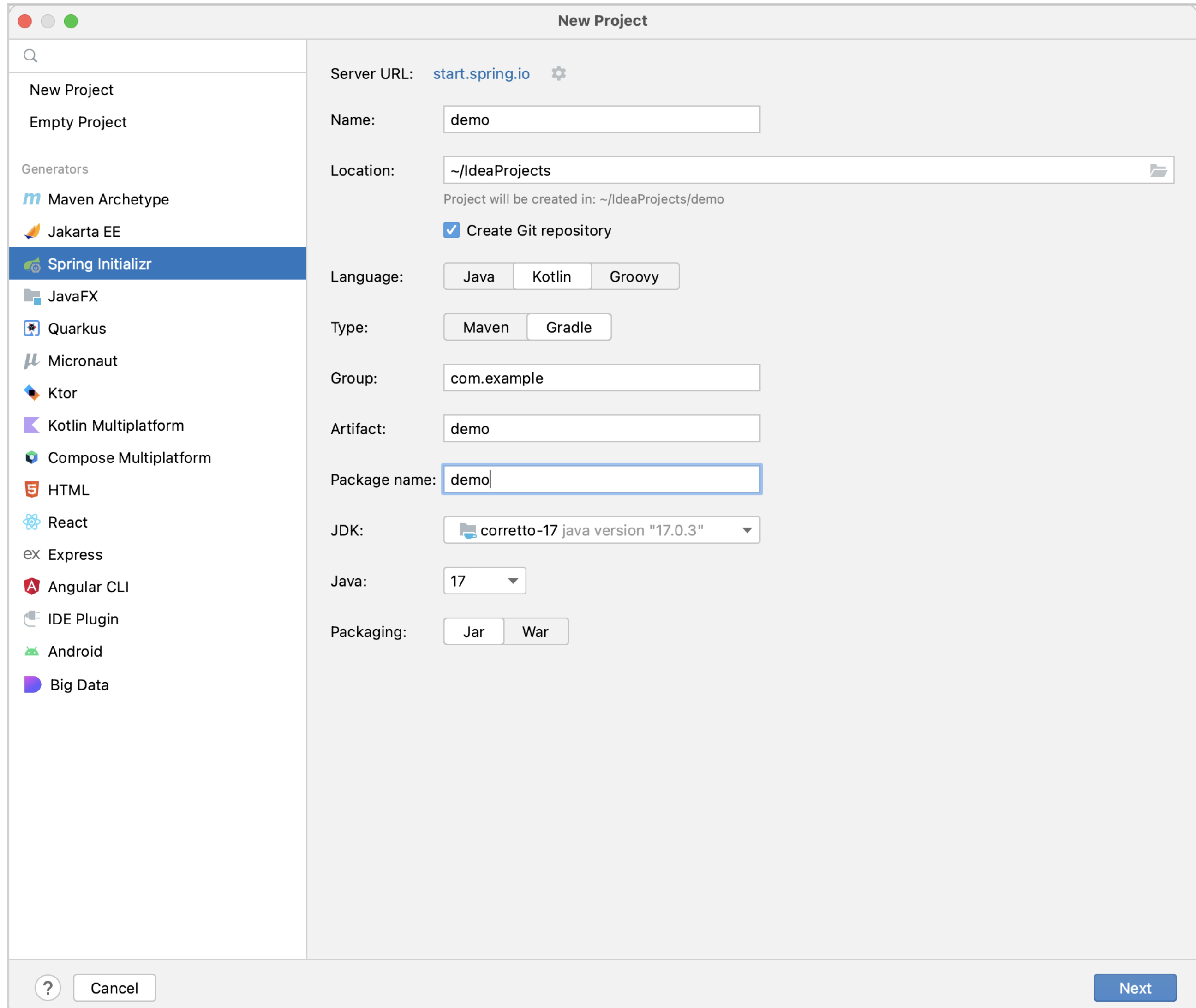
Create a new Spring Boot project with Kotlin by using the Project Wizard in IntelliJ IDEA Ultimate Edition:

> You can also create a new project using IntelliJ IDEA with the Spring Boot plugin.

1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select New Project | Spring Initializr.
3. Specify the following fields and options in the Project Wizard window:
   - Name: demo
   - Language: Kotlin
   - Build system: Gradle
   - JDK: Java 17 JDK

     > This tutorial uses Amazon Corretto version 18.

   - Java: 17

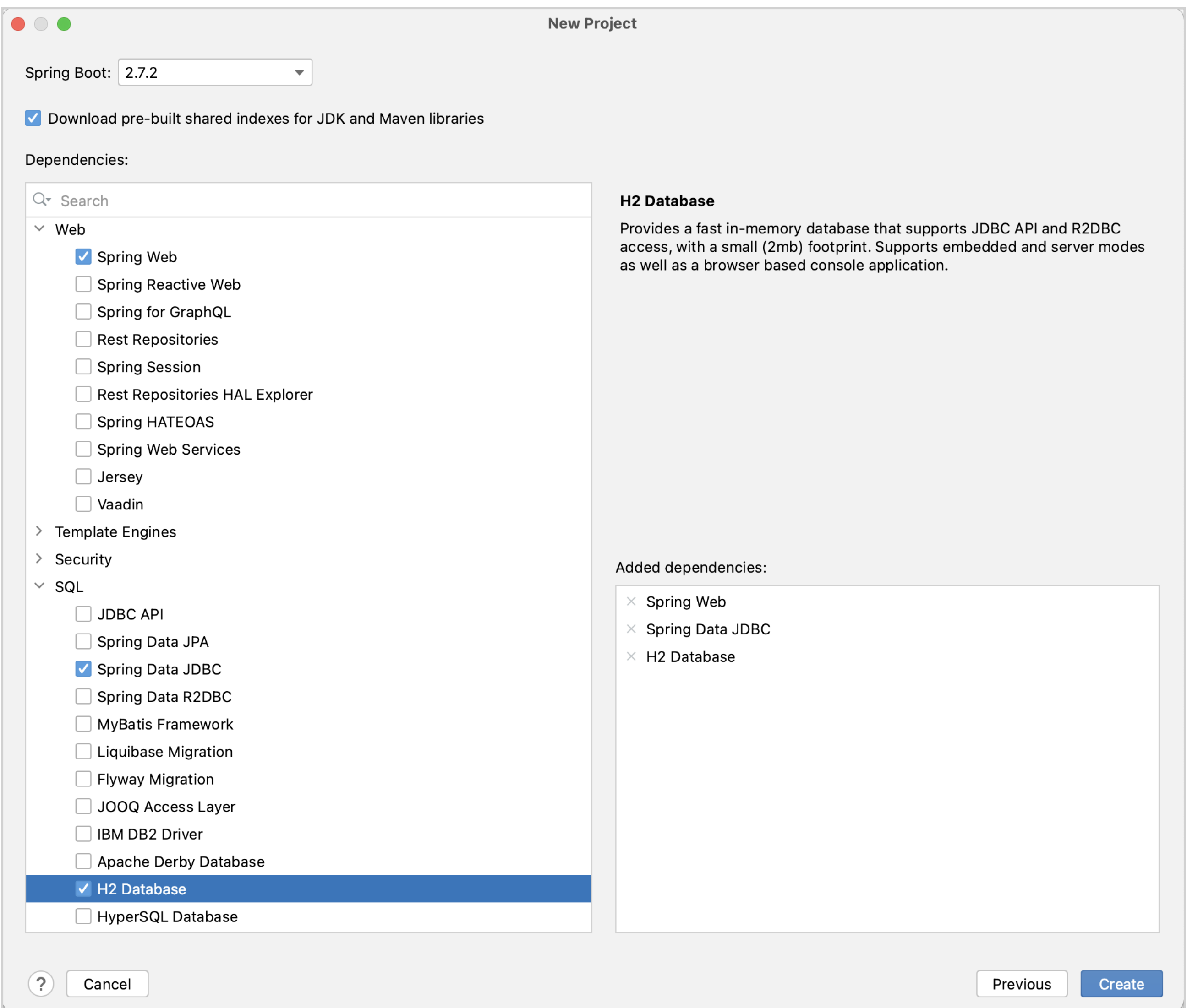


Create Spring Boot project

4. Ensure that you have specified all the fields and click Next.

5. Select the following dependencies that will be required for the tutorial:

- Web / Spring Web
- SQL / Spring Data JDBC
- SQL / H2 Database

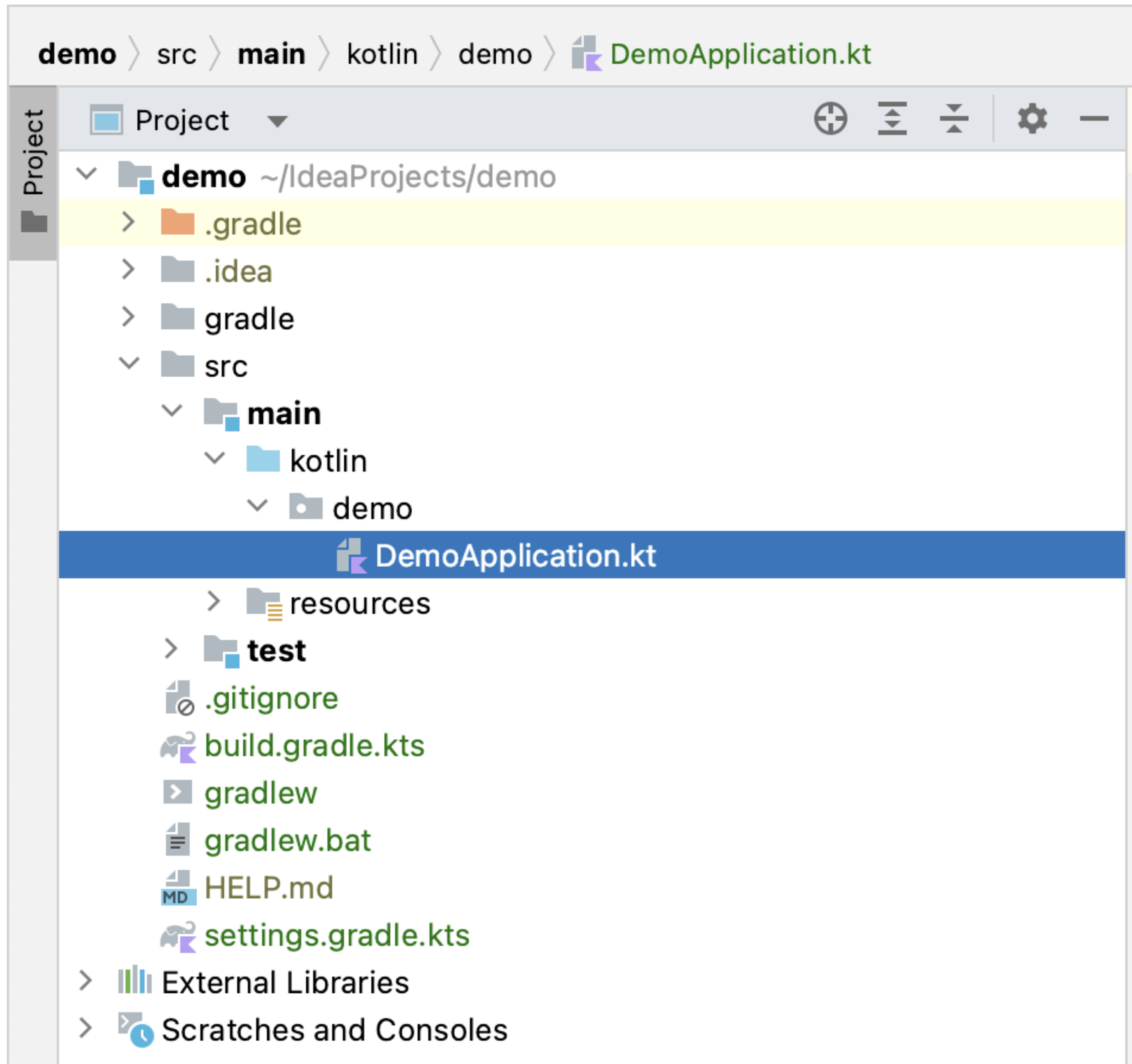


Set up Spring Boot project

6. Click Create to generate and set up the project.

   The IDE will generate and open the new project. It may take some time to download and import the project dependencies.

7. After this, you can observe the following structure in the Project view:



Set up Spring Boot project

The generated Gradle project corresponds to the Maven's standard directory layout:

- There are packages and classes under the main/kotlin folder that belong to the application.
- The entry point to the application is the main() method of the DemoApplication.kt file.

Explore the project Gradle build file

Open the build.gradle.kts file: it is the Gradle Kotlin build script, which contains a list of the dependencies required for the application.

The Gradle file is standard for Spring Boot, but it also contains necessary Kotlin dependencies, including the kotlin-spring Gradle plugin – kotlin("plugin.spring").

Here is the full script with the explanation of all parts and dependencies:

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile // For `KotlinCompile` task below

plugins {
    id("org.springframework.boot") version "2.7.1"
    id("io.spring.dependency-management") version "1.0.11.RELEASE"
    kotlin("jvm") version "1.9.0" // The version of Kotlin to use
    kotlin("plugin.spring") version "1.9.0" // The Kotlin Spring plugin
}
```

```
group = "com.example"
version = "0.0.1-SNAPSHOT"
java.sourceCompatibility = JavaVersion.VERSION_17

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-data-jdbc")
    implementation("org.springframework.boot:spring-boot-starter-web")
    implementation("com.fasterxml.jackson.module:jackson-module-kotlin") // Jackson extensions for Kotlin for working with JSON
    implementation("org.jetbrains.kotlin:kotlin-reflect") // Kotlin reflection library, required for working with Spring
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8") // Kotlin standard library
    runtimeOnly("com.h2database:h2")
    testImplementation("org.springframework.boot:spring-boot-starter-test")
}

tasks.withType<KotlinCompile> { // Settings for `KotlinCompile` tasks
    kotlinOptions { // Kotlin compiler options
        freeCompilerArgs = listOf("-Xjsr305=strict") // `-Xjsr305=strict` enables the strict mode for JSR-305 annotations
        jvmTarget = "17" // This option specifies the target version of the generated JVM bytecode
    }
}

tasks.withType<Test> {
    useJUnitPlatform()
}
```

As you can see, there are a few Kotlin-related artifacts added to the Gradle build file:

1. In the plugins block, there are two Kotlin artifacts:
   - kotlin("jvm") – the plugin defines the version of Kotlin to be used in the project
   - kotlin("plugin.spring") – Kotlin Spring compiler plugin for adding the open modifier to Kotlin classes in order to make them compatible with Spring Framework features
2. In the dependencies block, a few Kotlin-related modules listed:
   - com.fasterxml.jackson.module:jackson-module-kotlin – the module adds support for serialization and deserialization of Kotlin classes and data classes
   - org.jetbrains.kotlin:kotlin-reflect – Kotlin reflection library
   - org.jetbrains.kotlin:kotlin-stdlib-jdk8 – Kotlin standard library
3. After the dependencies section, you can see the KotlinComiple task configuration block. This is where you can add extra arguments to the compiler to enable or disable various language features.

Explore the generated Spring Boot application

Open the DemoApplication.kt file:

```
package demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

Declaring classes – class DemoApplication

Right after package declaration and import statements you can see the first class declaration, class DemoApplication.

In Kotlin, if a class doesn't include any members (properties or functions), you can omit the class body ({}) for good.

@SpringBootApplication annotation

@SpringBootApplication annotation is a convenience annotation in a Spring Boot application. It enables Spring Boot's auto-configuration, component scan, and be able to define an extra configuration on their "application class".

Program entry point – main()

The main() function is the entry point to the application.

It is declared as a top-level function outside the DemoApplication class. The main() function invokes the Spring's runApplication(&args) function to start the application with the Spring Framework.

Variable arguments – args: Array

If you check the declaration of the runApplication() function, you will see that the parameter of the function is marked with vararg modifier: vararg args: String. This means that you can pass a variable number of String arguments to the function.

The spread operator – (*args)

The args is a parameter to the main() function declared as an array of Strings. Since there is an array of strings, and you want to pass its content to the function, use the spread operator (prefix the array with a star sign *).

Create a controller

The application is ready to run, but let's update its logic first.

In the Spring application, a controller is used to handle the web requests. In the DemoApplication.kt file, create the MessageController class as follows:

```
@RestController
class MessageController {
    @GetMapping("/")
    fun index(@RequestParam("name") name: String) = "Hello, $name!"
}
```

@RestController annotation

You need to tell Spring that MessageController is a REST Controller, so you should mark it with the @RestController annotation.

This annotation means this class will be picked up by the component scan because it's in the same package as our DemoApplication class.

@GetMapping annotation

@GetMapping marks the functions of the REST controller that implement the endpoints corresponding to HTTP GET calls:

```
@GetMapping("/")
fun index(@RequestParam("name") name: String) = "Hello, $name!"
```

@RequestParam annotation

The function parameter name is marked with @RequestParam annotation. This annotation indicates that a method parameter should be bound to a web request parameter.

Hence, if you access the application at the root and supply a request parameter called "name", like /?name=<your-value>, the parameter value will be used as an argument for invoking the index() function.

Single-expression functions – index()

Since the index() function contains only one statement you can declare it as a single-expression function.

This means the curly braces can be omitted and the body is specified after the equals sign =.

Type inference for function return types

The index() function does not declare the return type explicitly. Instead, the compiler infers the return type by looking at the result of the statement on the right-hand side from the equals sign =.

The type of Hello, $name! expression is String, hence the return type of the function is also String.

String templates – $name

Hello, $name! expression is called a String template in Kotlin.

String templates are String literals that contain embedded expressions.

This is a convenient replacement for String concatenation operations.

These Spring annotations also require additional imports:

```
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController
```

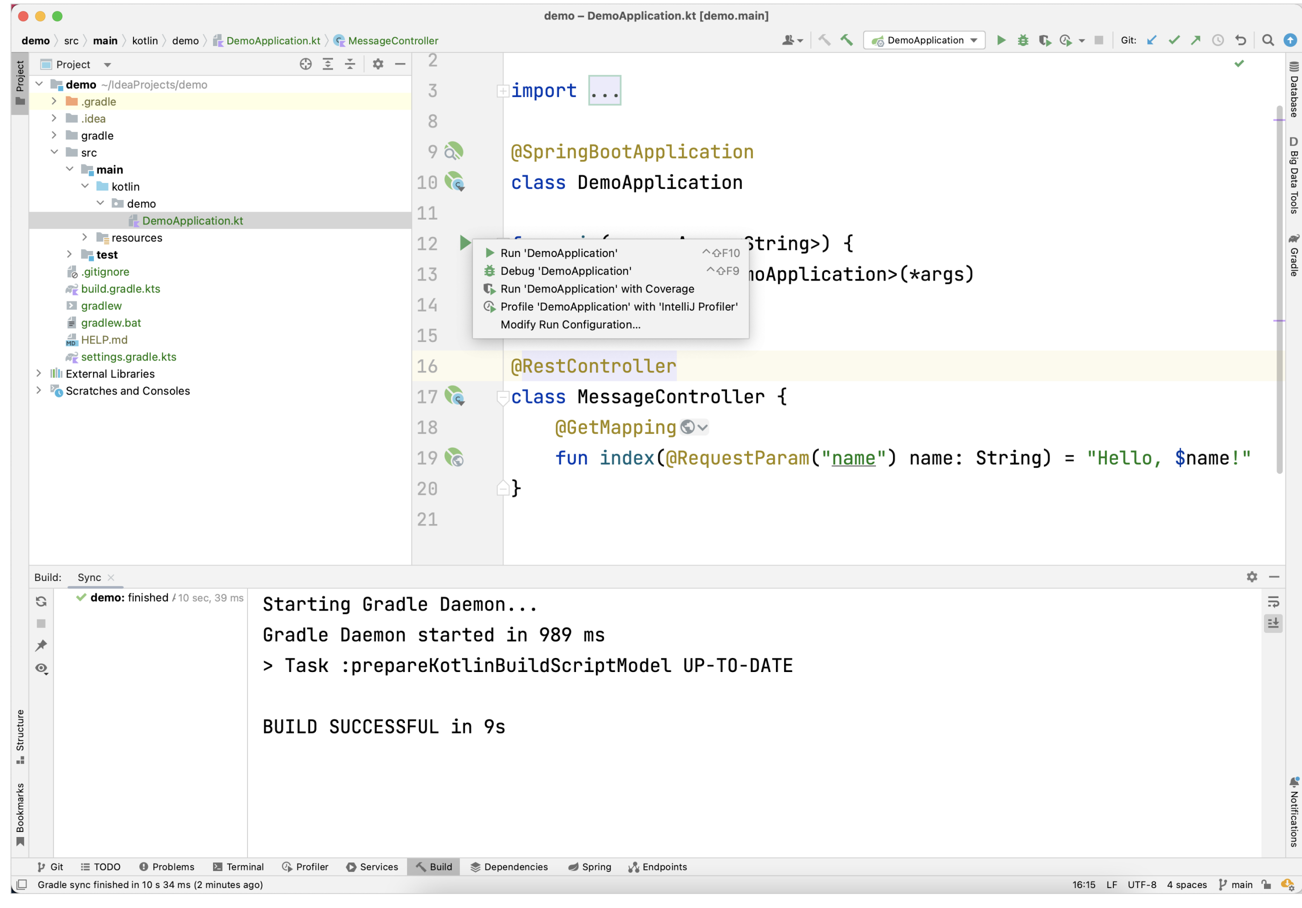
Here is a complete code of the DemoApplication.kt:

package demo import org.springframework.boot.autoconfigure.SpringBootApplication import org.springframework.boot.runApplication import org.springframework.web.bind.annotation.GetMapping import org.springframework.web.bind.annotation.RequestParam import org.springframework.web.bind.annotation.RestController @SpringBootApplication class DemoApplication fun main(args: Array<String>) { runApplication<DemoApplication>(*args) } @RestController class MessageController { @GetMapping("/") fun index(@RequestParam("name") name: String) = "Hello, $name!" }

Run the application

The Spring application is now ready to run:

1. Click the green Run icon in the gutter beside the main() method:



Run Spring Boot application

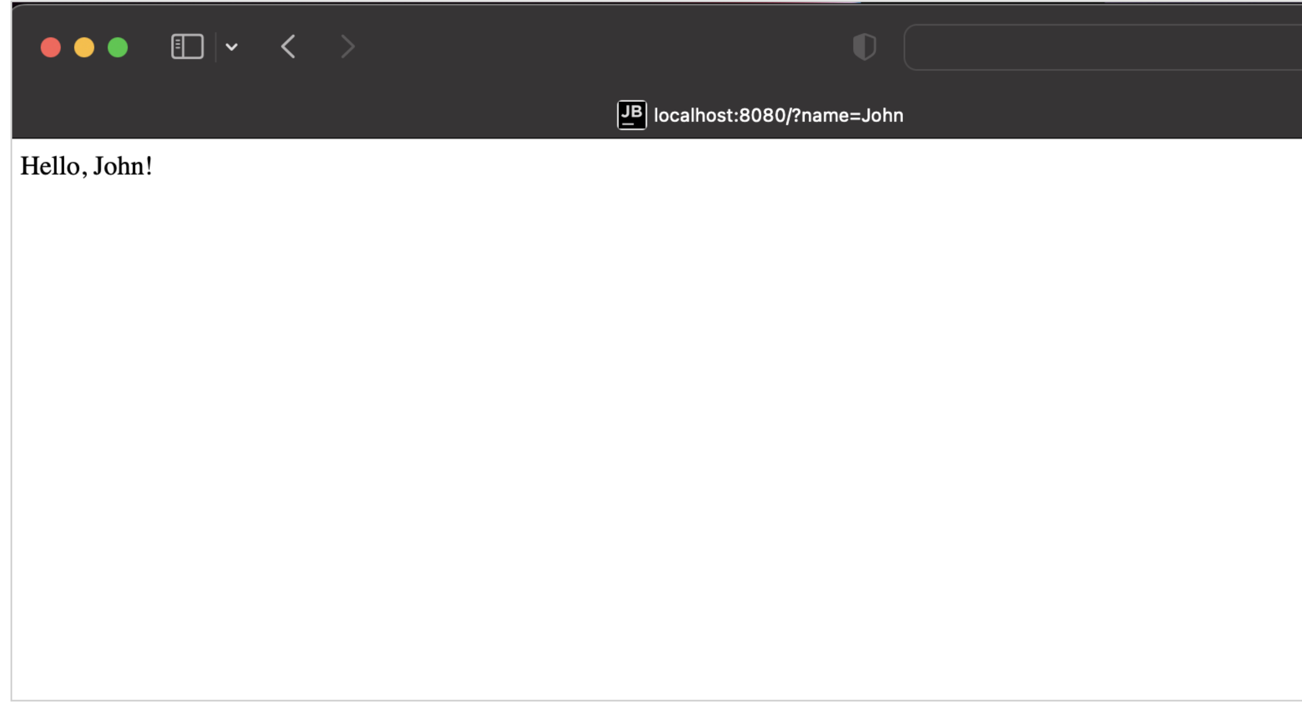
> You can also run the ./gradlew bootRun command in the terminal.

This starts the local server on your computer.

2. Once the application starts, open the following URL:

```
http://localhost:8080?name=John
```

You should see "Hello, John!" printed as a response:



Spring Application response

Next step

In the next part of the tutorial you'll learn about Kotlin data classes and how you can use them in your application.

Proceed to the next chapter

Get the Kotlin language map
Get your personal language map to help you navigate Kotlin features and track your progress in studying the language. We will also send you language tips and useful materials on using Kotlin with Spring.

Get the Kotlin language map

> You will need to share your email address on the next page to receive the materials.

Add a data class to Spring Boot project

In this part of the tutorial, you'll add some more functionality to the application and discover more Kotlin language features, such as data classes. It requires changing the MessageController class to respond with a JSON document containing a collection of serialized objects.

Update your application

1. In the DemoApplication.kt file, create a Message data class with two properties: id and text:

```
data class Message(val id: String?, val text: String)
```

Message class will be used for data transfer: a list of serialized Message objects will make up the JSON document that the controller is going to respond to the browser request.

Data classes – data class Message
The main purpose of data classes in Kotlin is to hold data. Such classes are marked with the data keyword, and some standard functionality and some utility functions are often mechanically derivable from the class structure.

In this example, you declared Message as a data class as its main purpose is to store the data.

val and var properties
Properties in Kotlin classes can be declared either as:

- mutable, using the var keyword
- read-only, using the val keyword

The Message class declares two properties using val keyword, the id and text. The compiler will automatically generate the getters for both of these properties. It will not be possible to reassign the values of these properties after an instance of the Message class is created.

Nullable types – String?
Kotlin provides built-in support for nullable types. In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-nullable references).
For example, a regular variable of type String cannot hold null. To allow nulls, you can declare a variable as a nullable string by writing String?.

The id property of the Message class is declared as a nullable type this time. Hence, it is possible to create an instance of Message class by passing null as a value for id:

```
Message(null, "Hello!")
```

2. In the same file, amend the index() function of a MessageController class to return a list of Message objects:

```
@RestController
class MessageController {
    @GetMapping("/")
    fun index() = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}
```

Collections – listOf()
The Kotlin Standard Library provides implementations for basic collection types: sets, lists, and maps.
A pair of interfaces represents each collection type:

- A read-only interface that provides operations for accessing collection elements.
- A mutable interface that extends the corresponding read-only interface with write operations: adding, removing, and updating its elements.

The corresponding factory functions are also provided by the Kotlin Standard Library to create instances of such collections.

In this tutorial, you use the listOf() function to create a list of Message objects. This is the factory function to create a read-only list of objects: you can't add or remove elements from the list.
If it is required to perform write operations on the list, call the mutableListOf() function to create a mutable list instance.

Trailing comma
A trailing comma is a comma symbol after the last item of a series of elements:

```
Message("3", "Privet!"),
```

This is a convenient feature of Kotlin syntax and is entirely optional – your code will still work without them.

In the example above, creating a list of Message objects includes the trailing comma after the last listOf() function argument.

The response from MessageController will now be a JSON document containing a collection of Message objects.

> Any controller in the Spring application renders JSON response by default if Jackson library is on the classpath. As you specified the spring-boot-starter-web dependency in the build.gradle.kts file, you received Jackson as a transitive dependency. Hence, the application responds with a JSON document if the endpoint returns a data structure that can be serialized to JSON.

Here is a complete code of the DemoApplication.kt:

```
package demo import org.springframework.boot.autoconfigure.SpringBootApplication import org.springframework.boot.runApplication import org.springframework.data.annotation.Id import org.springframework.web.bind.annotation.GetMapping import org.springframework.web.bind.annotation.RestController @SpringBootApplication class DemoApplication fun main(args: Array<String>) { runApplication<DemoApplication>(*args) } @RestController class MessageController { @GetMapping("/") fun index() = listOf( Message("1", "Hello!"), Message("2", "Bonjour!"), Message("3", "Privet!"), ) } data class Message(val id: String?, val text: String)
```

Run the application

The Spring application is ready to run:

1. Run the application again.
2. Once the application starts, open the following URL:

```
http://localhost:8080
```

You will see a page with a collection of messages in JSON format:

Run the application

Next step

In the next part of the tutorial, you'll add and configure a database to your project, and make HTTP requests.

Proceed to the next chapter

Get the Kotlin language map

Get your personal language map to help you navigate Kotlin features and track your progress in studying the language. We will also send you language tips and useful materials on using Kotlin with Spring.

Get the Kotlin Language Map →

Get the Kotlin language map

You will need to share your email address on the next page to receive the materials.

Add database support for Spring Boot project

In this part of the tutorial, you'll add and configure a database to your project using JDBC. In JVM applications, you use JDBC to interact with databases. For convenience, the Spring Framework provides the JdbcTemplate class that simplifies the use of JDBC and helps to avoid common errors.

Add database support

The common practice in Spring Framework based applications is to implement the database access logic within the so-called service layer – this is where business logic lives. In Spring, you should mark classes with the @Service annotation to imply that the class belongs to the service layer of the application. In this application, you will create the MessageService class for this purpose.

In the DemoApplication.kt file, create the MessageService class as follows:

```
import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate

@Service
class MessageService(val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun save(message: Message){
        db.update("insert into messages values ( ?, ? )",
           message.id, message.text)
    }
}
```

Constructor argument and dependency injection – (val db: JdbcTemplate)

A class in Kotlin can have a primary constructor and one or more secondary constructors. The primary constructor is a part of the class header, and it goes after the class name and optional type parameters. In our case, the constructor is (val db: JdbcTemplate).

val db: JdbcTemplate is the constructor's argument:

```
@Service
class MessageService(val db: JdbcTemplate)
```

Trailing lambda and SAM conversion

The findMessages() function calls the query() function of the JdbcTemplate class. The query() function takes two arguments: an SQL query as a String instance, and a callback that will map one object per row:

```
db.query("...", RowMapper { ... } )
```

The RowMapper interface declares only one method, so it is possible to implement it via lambda expression by omitting the name of the interface. The Kotlin

compiler knows the interface that the lambda expression needs to be converted to because you use it as a parameter for the function call. This is known as SAM conversion in Kotlin:

```
db.query("...", { ... } )
```

After the SAM conversion, the query function ends up with two arguments: a String at the first position, and a lambda expression at the last position. According to the Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses. Such syntax is also known as trailing lambda:

```
db.query("...") { ... }
```

Underscore for unused lambda argument
For a lambda with multiple parameters, you can use the underscore _ character to replace the names of the parameters you don't use.

Hence, the final syntax for query function call looks like this:

```
db.query("select * from messages") { response, _ ->
Message(response.getString("id"), response.getString("text"))
}
```

Update the MessageController class

Update MessageController to use the new MessageService class:

```
@RestController
class MessageController(val service: MessageService) {
    @GetMapping("/")
    fun index(): List<Message> = service.findMessages()

    @PostMapping("/")
    fun post(@RequestBody message: Message) {
       service.save(message)
    }
}
```

@PostMapping annotation
The method responsible for handling HTTP POST requests needs to be annotated with @PostMapping annotation. To be able to convert the JSON sent as HTTP Body content into an object, you need to use the @RequestBody annotation for the method argument. Thanks to having Jackson library in the classpath of the application, the conversion happens automatically.

Update the MessageService class

The id for Message class was declared as a nullable String:

```
data class Message(val id: String?, val text: String)
```

It would not be correct to store the null as an id value in the database though: you need to handle this situation gracefully.

Update your code to generate a new value when the id is null while storing the messages in the database:

```
@Service
class MessageService(val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun save(message: Message){
        val id = message.id ?: UUID.randomUUID().toString()
        db.update("insert into messages values ( ?, ? )",
```

```
                  id, message.text)
    }
}
```

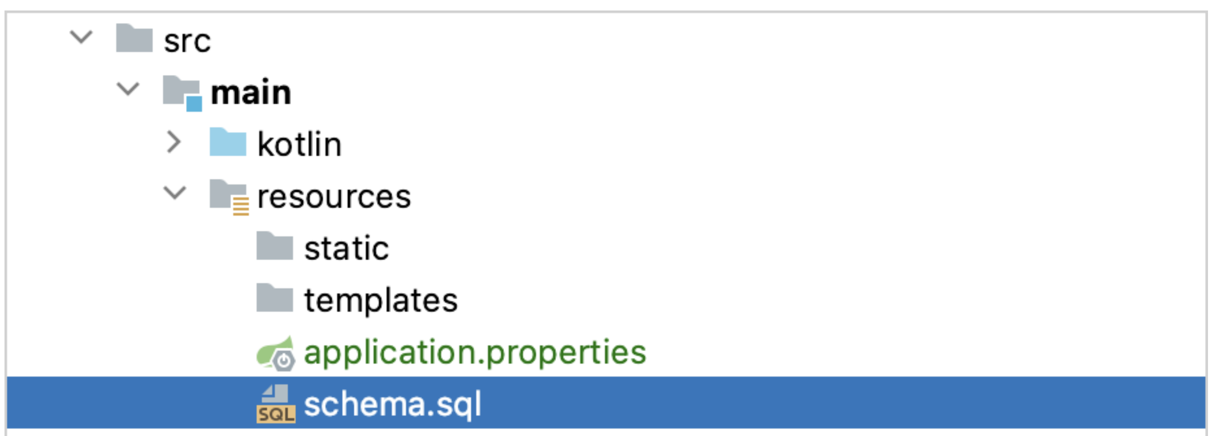
Elvis operator – ?:
The code message.id ?: UUID.randomUUID().toString() uses the Elvis operator (if-not-null-else shorthand) ?:. If the expression to the left of ?: is not null, the Elvis operator returns it; otherwise, it returns the expression to the right. Note that the expression on the right-hand side is evaluated only if the left-hand side is null.

The application code is ready to work with the database. It is now required to configure the data source.

Configure the database

Configure the database in the application:

1. Create schema.sql file in the src/main/resources directory. It will store the database object definitions:



Create database schema

2. Update the src/main/resources/schema.sql file with the following code:

```
CREATE TABLE IF NOT EXISTS messages (
id       VARCHAR(60)  PRIMARY KEY,
text     VARCHAR      NOT NULL
);
```

It creates the messages table with two columns: id and text. The table structure matches the structure of the Message class.

3. Open the application.properties file located in the src/main/resources folder and add the following application properties:

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./data/testdb
spring.datasource.username=name
spring.datasource.password=password
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.mode=always
```

These settings enable the database for the Spring Boot application.
See the full list of common application properties in the Spring documentation.

Add messages to database via HTTP request

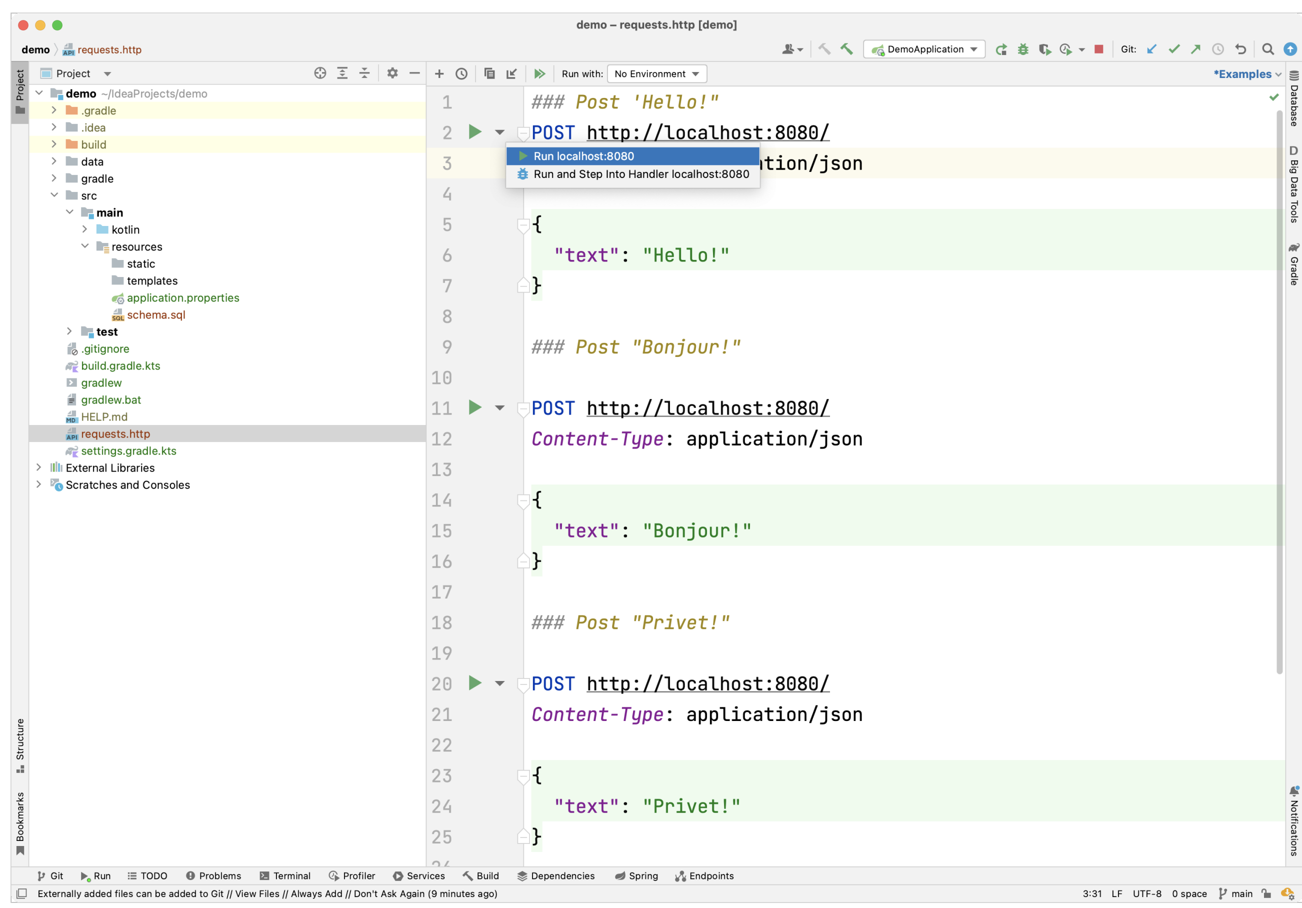
You should use an HTTP client to work with previously created endpoints. In IntelliJ IDEA, use the embedded HTTP client:

1. Run the application. Once the application is up and running, you can execute POST requests to store messages in the database. Create the requests.http file and add the following HTTP requests:

```
### Post "Hello!"
```

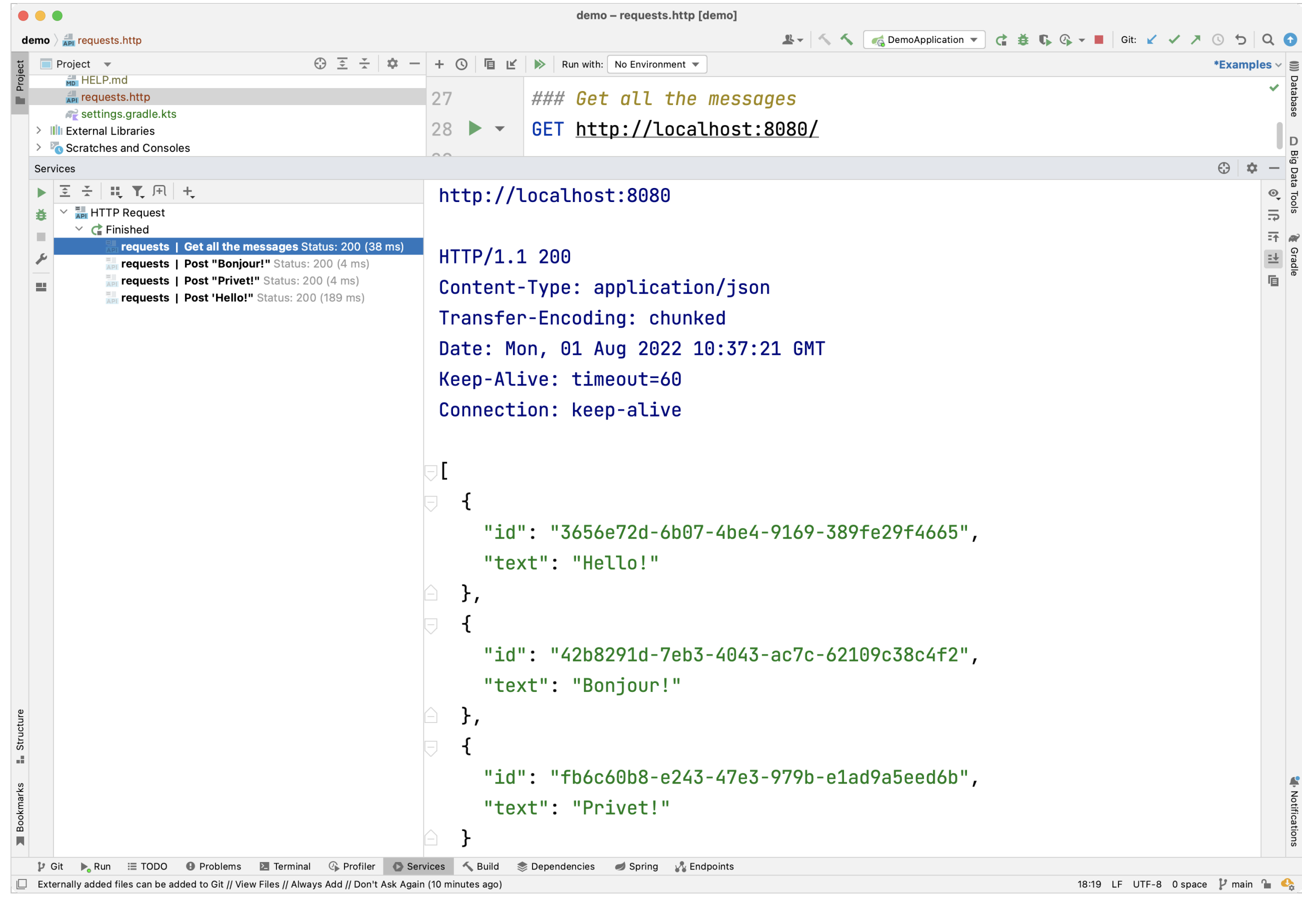
```
POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Hello!"
}

### Post "Bonjour!"

POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Bonjour!"
}

### Post "Privet!"

POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Privet!"
}

### Get all the messages
GET http://localhost:8080/
```

2. Execute all POST requests. Use the green Run icon in the gutter next to the request declaration. These requests write the text messages to the database:



Execute POST request

3. Execute the GET request and see the result in the Run tool window:



Execute GET requests

Alternative way to execute requests

You can also use any other HTTP client or the cURL command-line tool. For example, run the following commands in the terminal to get the same result:

```
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{ \"text\": \"Hello!\" }"

curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{ \"text\": \"Bonjour!\" }"

curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{ \"text\": \"Privet!\" }"

curl -X GET --location "http://localhost:8080"
```

Retrieve messages by id

Extend the functionality of the application to retrieve the individual messages by id.

1. In the MessageService class, add the new function findMessageById(id: String) to retrieve the individual messages by id:

```
import org.springframework.jdbc.core.query

@Service
class MessageService(val db: JdbcTemplate) {

    fun findMessages(): List<Message> = db.query("select * from messages") { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun findMessageById(id: String): List<Message> = db.query("select * from messages where id = ?", id) { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun save(message: Message) {
```

```
        val id = message.id ?: UUID.randomUUID().toString()
        db.update("insert into messages values ( ?, ? )",
                  id, message.text)
    }
}
```

The query() function that is used to fetch the message by its id is a Kotlin extension function provided by the Spring Framework and requires an additional import as in the code above.

2. Add the new index(...) function with the id parameter to the MessageController class:

```
@RestController
class MessageController(val service: MessageService) {
    @GetMapping("/")
    fun index(): List<Message> = service.findMessages()

    @GetMapping("/{id}")
    fun index(@PathVariable id: String): List<Message> =
        service.findMessageById(id)

    @PostMapping("/")
    fun post(@RequestBody message: Message) {
        service.save(message)
    }
}
```

Retrieving a value from the context path
The message id is retrieved from the context path by the Spring Framework as you annotated the new function by @GetMapping("/{id}"). By annotating the function argument with @PathVariable, you tell the framework to use the retrieved value as a function argument. The new function makes a call to MessageService to retrieve the individual message by its id.

vararg argument position in the parameter list
The query() function takes three arguments:

- SQL query string that requires a parameter to run
- `id`, which is a parameter of type String
- RowMapper instance is implemented by a lambda expression

The second parameter for the query() function is declared as a variable argument (vararg). In Kotlin, the position of the variable arguments parameter is not required to be the last in the parameters list.

Run the application

The Spring application is ready to run:

1. Run the application again.
2. Open the requests.http file and add the new GET request:

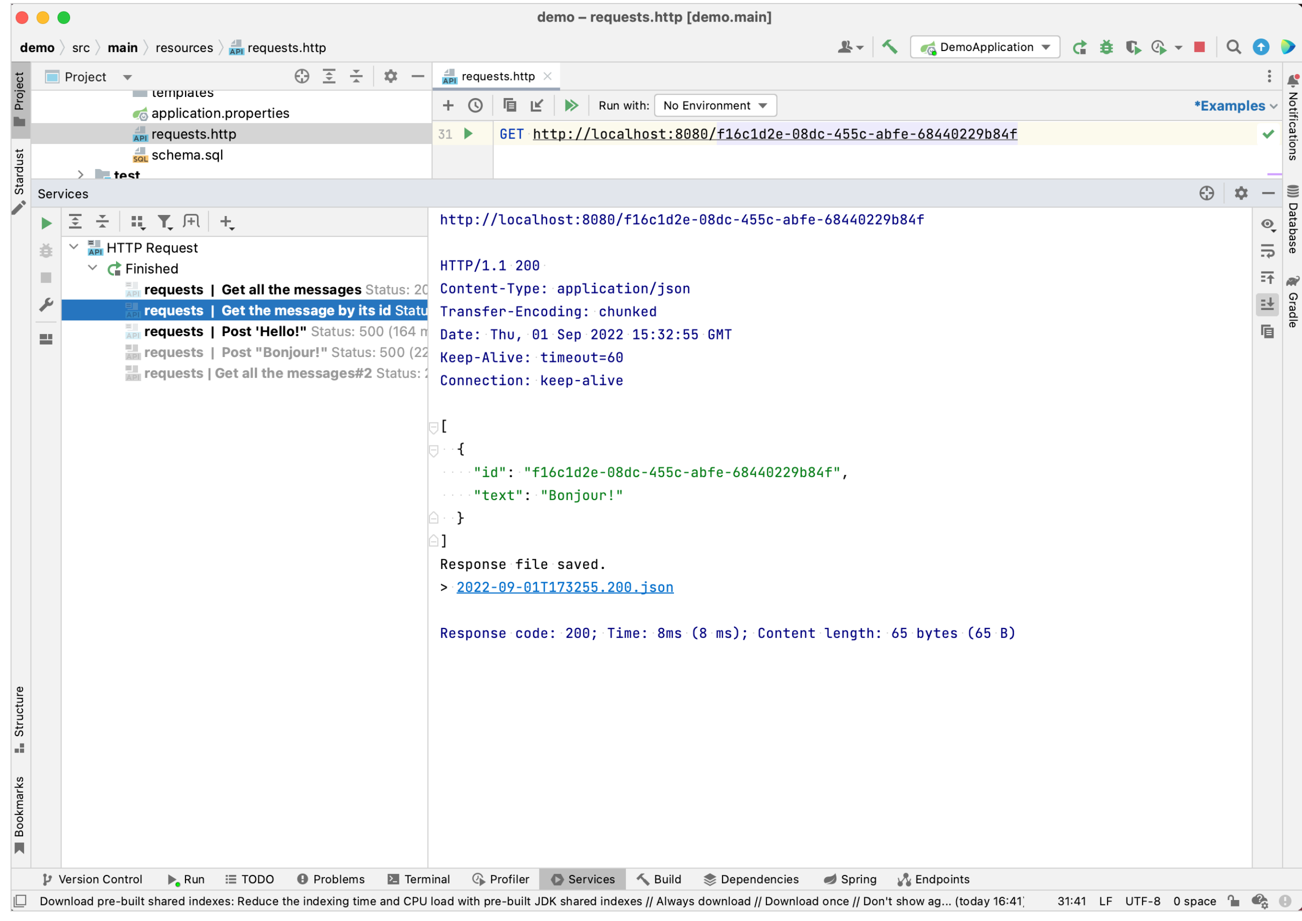
```
### Get the message by its id
GET http://localhost:8080/id
```

3. Execute the GET request to retrieve all the messages from the database.
4. In the Run tool window copy one of the ids and add it to the request, like this:

```
### Get the message by its id
GET http://localhost:8080/f16c1d2e-08dc-455c-abfe-68440229b84f
```

Put your message id instead of the mentioned above.

5. Execute the GET request and see the result in the Run tool window:



Retrieve message by its id

Next step

The final step shows you how to use more popular connection to database using Spring Data.

Proceed to the next chapter

Get the Kotlin language map

Get your personal language map to help you navigate Kotlin features and track your progress in studying the language. We will also send you language tips and useful materials on using Kotlin with Spring.

Get the Kotlin language map

> You will need to share your email address on the next page to receive the materials.

Use Spring Data CrudRepository for database access

In this part, you will migrate the service layer to use the Spring Data CrudRepository instead of JdbcTemplate for database access. CrudRepository is a Spring Data interface for generic CRUD operations on a repository of a specific type. It provides several methods out of the box for interacting with a database.

Update your application

First, you need to adjust the Message class for work with the CrudRepository API:

1. Add the @Table annotation to the Message class to declare mapping to a database table.
   Add the @Id annotation before the id field.

   > These annotations also require additional imports.

   ```
   import org.springframework.data.annotation.Id
   import org.springframework.data.relational.core.mapping.Table

   @Table("MESSAGES")
   data class Message(@Id var id: String?, val text: String)
   ```

   Besides adding the annotations, you also need to make the id mutable (var) for the reasons of how CrudRepository works when inserting the new objects to the database.

2. Declare an interface for the CrudRepository that will work with the Message data class:

   ```
   import org.springframework.data.repository.CrudRepository

   interface MessageRepository : CrudRepository<Message, String>
   ```

3. Update the MessageService class. It will now call to the MessageRepository instead of executing SQL queries:

   ```
   @Service
   class MessageService(val db: MessageRepository) {
       fun findMessages(): List<Message> = db.findAll().toList()

       fun findMessageById(id: String): List<Message> = db.findById(id).toList()

       fun save(message: Message) {
           db.save(message)
       }

       fun <T : Any> Optional<out T>.toList(): List<T> =
           if (isPresent) listOf(get()) else emptyList()
   }
   ```

   Extension functions
   The return type of the findById() function in the CrudRepository interface is an instance of the Optional class. However, it would be convenient to return a List with a single message for consistency. For that, you need to unwrap the Optional value if it's present, and return a list with the value. This can be implemented as an extension function to the Optional type.

   In the code, Optional<out T>.toList(), toList() is the extension function for Optional. Extension functions allow you to write additional functions to any classes, which is especially useful when you want to extend functionality of some library class.

   CrudRepository save() function
   This function works with an assumption that the new object doesn't have an id in the database. Hence, the id should be null for insertion.

   If the id isn't null, CrudRepository assumes that the object already exists in the database and this is an update operation as opposed to an insert operation. After the insert operation, the id will be generated by the data store and assigned back to the Message instance. This is why the id property should be declared using the var keyword.

4. Update the messages table definition to generate the ids for the inserted objects. Since id is a string, you can use the RANDOM_UUID() function to generate the id value by default:

   ```
   CREATE TABLE messages (
      id                     VARCHAR(60)  DEFAULT RANDOM_UUID() PRIMARY KEY,
      text                   VARCHAR      NOT NULL
   ```

```
    );
```

Run the application

The application is ready to run again. By replacing the JdbcTemplate with CrudRepository, the functionality didn't change hence the application should work the same way as previously.

Next step

Get your personal language map to help you navigate Kotlin features and track your progress in studying the language. We will also send you language tips and useful materials on using Kotlin with Spring.

Get the Kotlin Language Map →

Get the Kotlin language map

> You will need to share your email address on the next page to receive the materials.

Test code using JUnit in JVM – tutorial

This tutorial will show you how to write a simple unit test and run it with the Gradle build tool.

The example in the tutorial has the kotlin.test library under the hood and runs the test using JUnit.

To get started, first download and install the latest version of IntelliJ IDEA.

Add dependencies

1. Open a Kotlin project in IntelliJ IDEA. If you don't already have a project, create one.

   > Specify JUnit 5 as your test framework when creating your project.

2. Open the build.gradle(.kts) file and add the following dependency to the Gradle configuration. This dependency will allow you to work with kotlin.test and JUnit:

   Kotlin

   ```
   dependencies {
       // Other dependencies.
       testImplementation(kotlin("test"))
   }
   ```

   Groovy

   ```
   dependencies {
       // Other dependencies.
       testImplementation 'org.jetbrains.kotlin:kotlin-test'
   }
   ```

3. Add the test task to the build.gradle(.kts) file:

Kotlin

```
tasks.test {
    useJUnitPlatform()
}
```

Groovy

```
test {
    useJUnitPlatform()
}
```

> If you created the project using the New Project wizard, the task will be added automatically.

Add the code to test it

1. Open the main.kt file in src/main/kotlin.

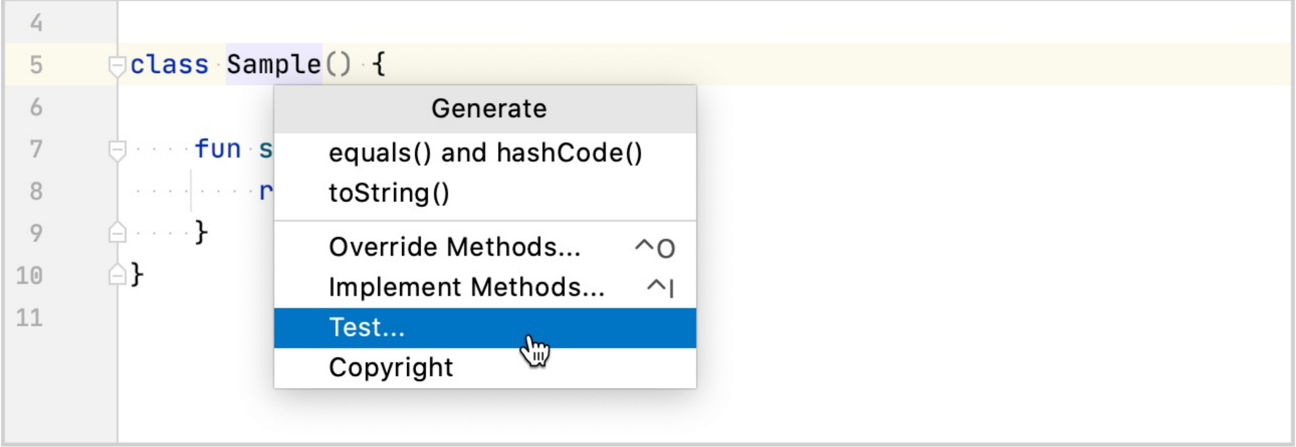
   The src directory contains Kotlin source files and resources. The main.kt file contains sample code that will print Hello, World!.

2. Create the Sample class with the sum() function that adds two integers together:

```
class Sample() {

    fun sum(a: Int, b: Int): Int {
        return a + b
    }
}
```

Create a test

1. In IntelliJ IDEA, select Code | Generate | Test... for the Sample class.



Create a test

2. Specify the name of the test class. For example, SampleTest.

   IntelliJ IDEA creates the SampleTest.kt file in the test directory. This directory contains Kotlin test source files and resources.

> You can also manually create a *.kt file for tests in src/test/kotlin.

3. Add the test code for the sum() function in SampleTest.kt:

   - Define the test testSum() function using the @Test annotation.
   - Check that the sum() function returns the expected value by using the assertEquals() function.

```kotlin
import kotlin.test.Test
import kotlin.test.assertEquals

internal class SampleTest {

    private val testSample: Sample = Sample()

    @Test
    fun testSum() {
        val expected = 42
        assertEquals(expected, testSample.sum(40, 2))
    }
}
```

Run a test

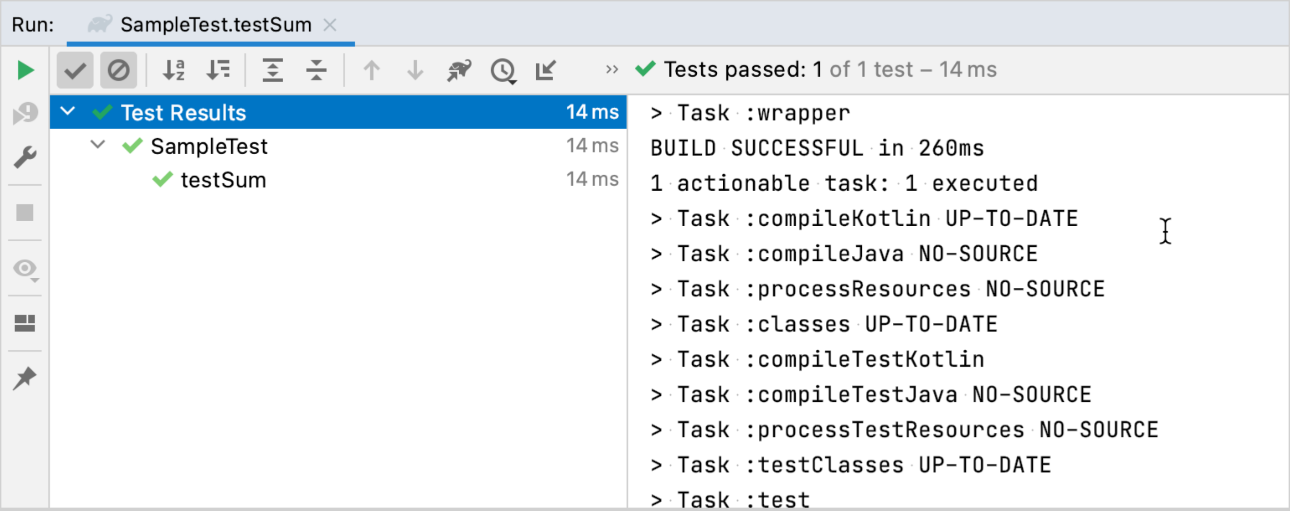
1. Run the test using the gutter icon.

```
7
8      @Test
9      fun testSum() {
10         val expected = 42
11         assertEquals(expected, testSample.sum( a: 40,  b: 2))
12     }
13 }
```

Run the test

> You can also run all project tests via the command-line interface using the ./gradlew check command.

2. Check the result in the Run tool window:



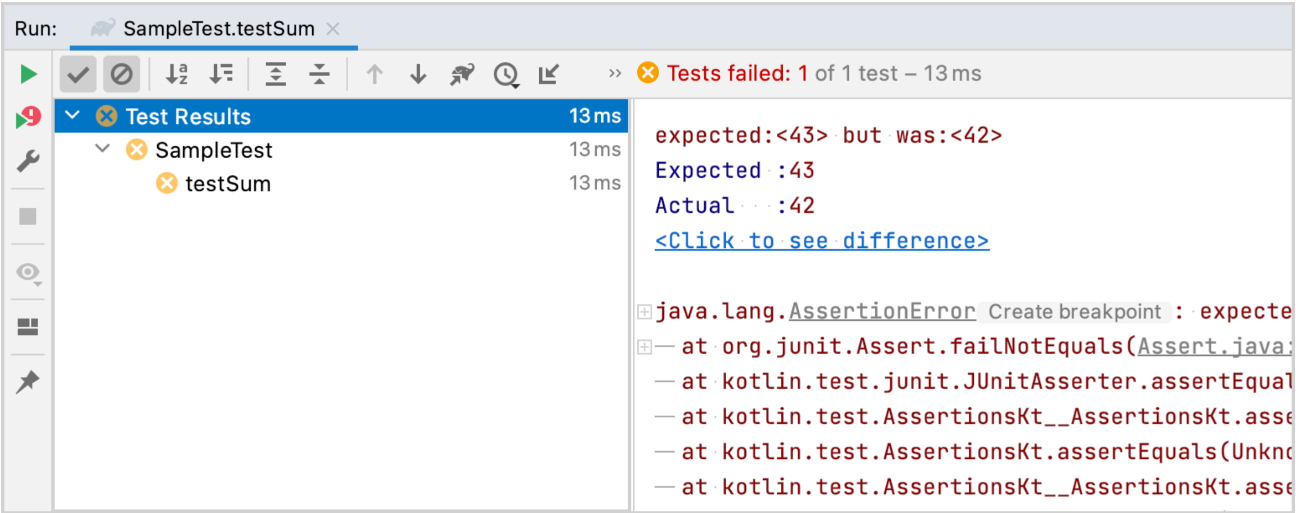
Check the test result. The test passed successfully

The test function was executed successfully.

3. Make sure that the test works correctly by changing the expected variable value to 43:

```kotlin
@Test
fun testSum() {
    val expected = 43
    assertEquals(expected, classForTesting.sum(40, 2))
}
```

4. Run the test again and check the result:



Check the test result. The test has been failed

The test execution failed.

What's next

Once you've finished your first test, you can:

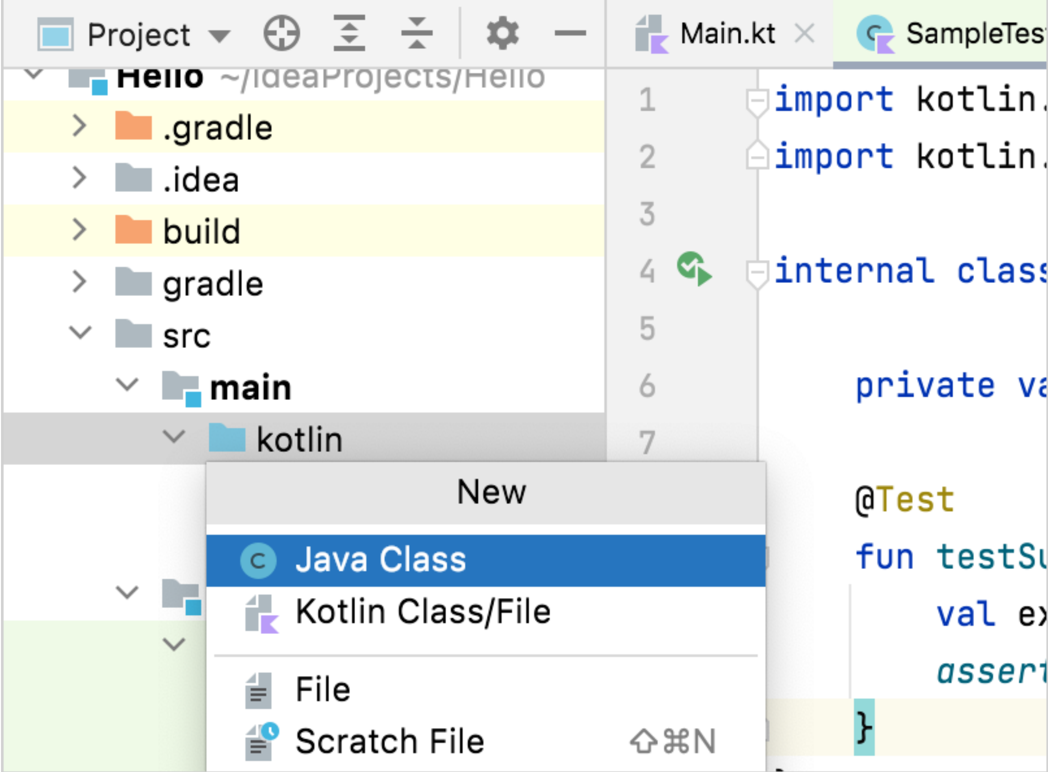
- Try to write another test using other kotlin.test functions. For example, you could use the assertNotEquals() function.
- Create your first application with Kotlin and Spring Boot.
- Watch these video tutorials on YouTube, which demonstrate how to use Spring Boot with Kotlin and JUnit 5.

Mixing Java and Kotlin in one project – tutorial

Kotlin provides the first-class interoperability with Java, and modern IDEs make it even better. In this tutorial, you'll learn how to use both Kotlin and Java sources in the same project in IntelliJ IDEA. To learn how to start a new Kotlin project in IntelliJ IDEA, see Getting started with IntelliJ IDEA.

Adding Java source code to an existing Kotlin project

Adding Java classes to a Kotlin project is pretty straightforward. All you need to do is create a new Java file. Select a directory or a package inside your project and go to File | New | Java Class or use the Alt + Insert/Cmd + N shortcut.



Add new Java class

If you already have the Java classes, you can just copy them to the project directories.

You can now consume the Java class from Kotlin or vice versa without any further actions.

For example, adding the following Java class:

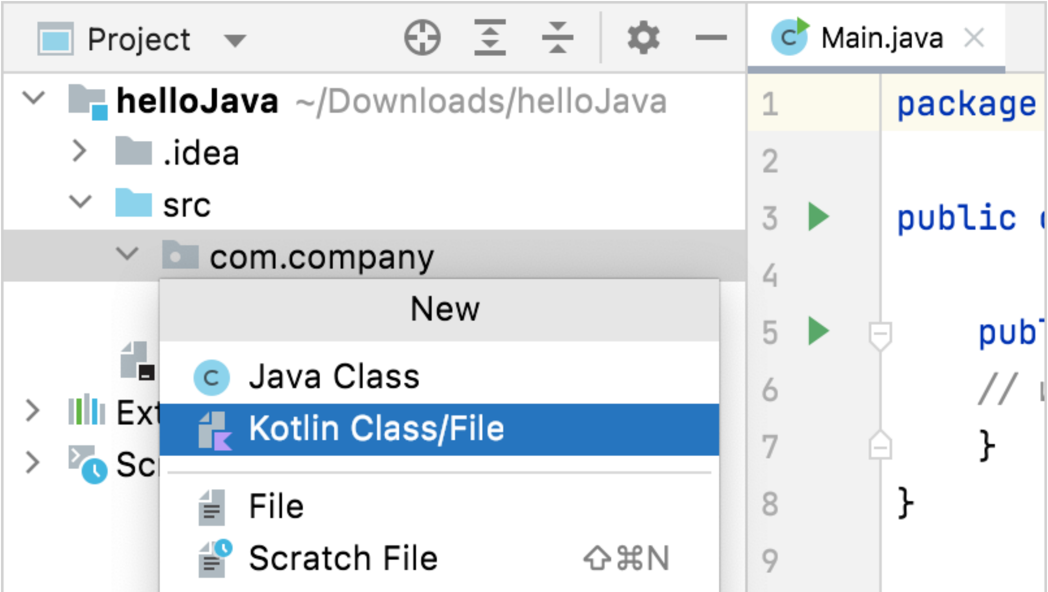
```
public class Customer {

    private String name;

    public Customer(String s){
        name = s;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void placeOrder() {
        System.out.println("A new order is placed by " + name);
    }
}
```

lets you call it from Kotlin like any other type in Kotlin.

```
val customer = Customer("Phase")
println(customer.name)
println(customer.placeOrder())
```

Adding Kotlin source code to an existing Java project

Adding a Kotlin file to an existing Java project is pretty much the same.



Add new Kotlin file class

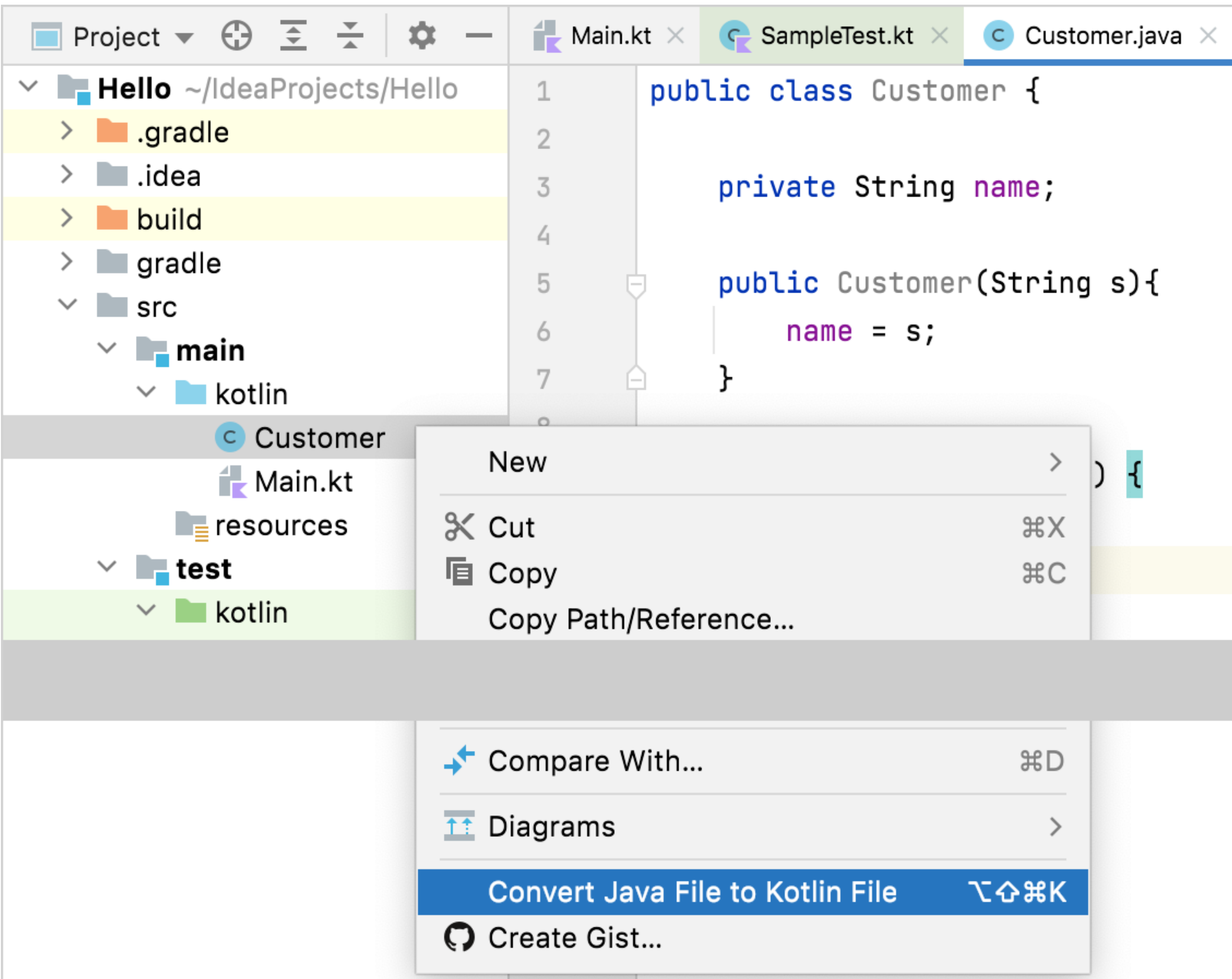
If this is the first time you're adding a Kotlin file to this project, IntelliJ IDEA will automatically add the required Kotlin runtime.

Bundling Kotlin runtime

You can also open the Kotlin runtime configuration manually from Tools | Kotlin | Configure Kotlin in Project.

Converting an existing Java file to Kotlin with J2K

The Kotlin plugin also bundles a Java to Kotlin converter (J2K) that automatically converts Java files to Kotlin. To use J2K on a file, click Convert Java File to Kotlin File in its context menu or in the Code menu of IntelliJ IDEA.



Convert Java to Kotlin

While the converter is not fool-proof, it does a pretty decent job of converting most boilerplate code from Java to Kotlin. Some manual tweaking however is sometimes required.

Using Java records in Kotlin

Records are classes in Java for storing immutable data. Records carry a fixed set of values – the records components. They have a concise syntax in Java and save you from having to write boilerplate code:

```
// Java
public record Person (String name, int age) {}
```

The compiler automatically generates a final class inherited from java.lang.Record with the following members:

- a private final field for each record component
- a public constructor with parameters for all fields
- a set of methods to implement structural equality: equals(), hashCode(), toString()
- a public method for reading each record component

Records are very similar to Kotlin data classes.

Using Java records from Kotlin code

You can use record classes with components that are declared in Java the same way you would use classes with properties in Kotlin. To access the record component, just use its name like you do for Kotlin properties:

```
val newPerson = Person("Kotlin", 10)
val firstName = newPerson.name
```

Declare records in Kotlin

Kotlin supports record declaration only for data classes, and the data class must meet the requirements.

To declare a record class in Kotlin, use the @JvmRecord annotation:

> Applying @JvmRecord to an existing class is not a binary compatible change. It alters the naming convention of the class property accessors.

```
@JvmRecord
data class Person(val name: String, val age: Int)
```

This JVM-specific annotation enables generating:

- the record components corresponding to the class properties in the class file
- the property accessor methods named according to the Java record naming convention

The data class provides equals(), hashCode(), and toString() method implementations.

Requirements

To declare a data class with the @JvmRecord annotation, it must meet the following requirements:

- The class must be in a module that targets JVM 16 bytecode (or 15 if the -Xjvm-enable-preview compiler option is enabled).
- The class cannot explicitly inherit any other class (including Any) because all JVM records implicitly inherit java.lang.Record. However, the class can implement interfaces.
- The class cannot declare any properties with backing fields – except those initialized from the corresponding primary constructor parameters.
- The class cannot declare any mutable properties with backing fields.
- The class cannot be local.
- The primary constructor of the class must be as visible as the class itself.

Enabling JVM records

JVM records require the 16 target version or higher of the generated JVM bytecode.

To specify it explicitly, use the jvmTarget compiler option in Gradle or Maven.

Further discussion

See this language proposal for JVM records for further technical details and discussion.

Strings in Java and Kotlin

This guide contains examples of how to perform typical tasks with strings in Java and Kotlin. It will help you migrate from Java to Kotlin and write your code in the authentically Kotlin way.

Concatenate strings

In Java, you can do this in the following way:

```
// Java
String name = "Joe";
System.out.println("Hello, " + name);
System.out.println("Your name is " + name.length() + " characters long");
```

In Kotlin, use the dollar sign ($) before the variable name to interpolate the value of this variable into your string:

```
fun main() {
//sampleStart
    // Kotlin
    val name = "Joe"
    println("Hello, $name")
    println("Your name is ${name.length} characters long")
//sampleEnd
}
```

You can interpolate the value of a complicated expression by surrounding it with curly braces, like in ${name.length}. See string templates for more information.

Build a string

In Java, you can use the StringBuilder:

```
// Java
StringBuilder countDown = new StringBuilder();
for (int i = 5; i > 0; i--) {
    countDown.append(i);
    countDown.append("\n");
}
System.out.println(countDown);
```

In Kotlin, use buildString() – an inline function that takes logic to construct a string as a lambda argument:

```
fun main() {
//sampleStart
       // Kotlin
       val countDown = buildString {
           for (i in 5 downTo 1) {
               append(i)
               appendLine()
           }
       }
       println(countDown)
//sampleEnd
}
```

Under the hood, the buildString uses the same StringBuilder class as in Java, and you access it via an implicit this inside the lambda.

Learn more about lambda coding conventions.

Create a string from collection items

In Java, you use the Stream API to filter, map, and then collect the items:

```
// Java
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
String invertedOddNumbers = numbers
        .stream()
        .filter(it -> it % 2 != 0)
        .map(it -> -it)
        .map(Object::toString)
        .collect(Collectors.joining("; "));
System.out.println(invertedOddNumbers);
```

In Kotlin, use the joinToString() function, which Kotlin defines for every List:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf(1, 2, 3, 4, 5, 6)
    val invertedOddNumbers = numbers
        .filter { it % 2 != 0 }
        .joinToString(separator = ";") {"${-it}"}
    println(invertedOddNumbers)
//sampleEnd
}
```

In Java, if you want spaces between your delimiters and following items, you need to add a space to the delimiter explicitly.

Learn more about joinToString() usage.

Set default value if the string is blank

In Java, you can use the ternary operator:

```
// Java
public void defaultValueIfStringIsBlank() {
    String nameValue = getName();
    String name = nameValue.isBlank() ? "John Doe" : nameValue;
    System.out.println(name);
}

public String getName() {
    Random rand = new Random();
    return rand.nextBoolean() ? "" : "David";
}
```

Kotlin provides the inline function ifBlank() that accepts the default value as an argument:

```
// Kotlin
import kotlin.random.Random

//sampleStart
fun main() {
    val name = getName().ifBlank { "John Doe" }
    println(name)
}

fun getName(): String =
    if (Random.nextBoolean()) "" else "David"
//sampleEnd
```

Replace characters at the beginning and end of a string

In Java, you can use the replaceAll() function. The replaceAll() function in this case accepts regular expressions ^## and ##$, which define strings starting and ending with ## respectively:

```
// Java
String input = "##place##holder##";
String result = input.replaceAll("^##|##$", "");
System.out.println(result);
```

In Kotlin, use the removeSurrounding() function with the string delimiter ##:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "##place##holder##"
    val result = input.removeSurrounding("##")
```

```
    println(result)
//sampleEnd
}
```

Replace occurrences

In Java, you can use the Pattern and the Matcher classes, for example, to obfuscate some data:

```
// Java
String input = "login: Pokemon5, password: 1q2w3e4r5t";
Pattern pattern = Pattern.compile("\\w*\\d+\\w*");
Matcher matcher = pattern.matcher(input);
String replacementResult = matcher.replaceAll(it -> "xxx");
System.out.println("Initial input: '" + input + "'");
System.out.println("Anonymized input: '" + replacementResult + "'");
```

In Kotlin, you use the Regex class that simplifies working with regular expressions. Additionally, use raw strings to simplify a regex pattern by reducing the count of backslashes:

```
fun main() {
//sampleStart
    // Kotlin
    val regex = Regex("""\w*\d+\w*""") // raw string
    val input = "login: Pokemon5, password: 1q2w3e4r5t"
    val replacementResult = regex.replace(input, replacement = "xxx")
    println("Initial input: '$input'")
    println("Anonymized input: '$replacementResult'")
//sampleEnd
}
```

Split a string

In Java, to split a string with the period character (.), you need to use shielding (\\). This happens because the split() function of the String class accepts a regular expression as an argument:

```
// Java
System.out.println(Arrays.toString("Sometimes.text.should.be.split".split("\\.")));
```

In Kotlin, use the Kotlin function split(), which accepts varargs of delimiters as input parameters:

```
fun main() {
//sampleStart
    // Kotlin
    println("Sometimes.text.should.be.split".split("."))
//sampleEnd
}
```

If you need to split with a regular expression, use the overloaded split() version that accepts the Regex as a parameter.

Take a substring

In Java, you can use the substring() function, which accepts an inclusive beginning index of a character to start taking the substring from. To take a substring after this character, you need to increment the index:

```
// Java
String input = "What is the answer to the Ultimate Question of Life, the Universe, and Everything? 42";
String answer = input.substring(input.indexOf("?") + 1);
System.out.println(answer);
```

In Kotlin, you use the substringAfter() function and don't need to calculate the index of the character you want to take a substring after:

```
fun main() {
//sampleStart
```

```
    // Kotlin
    val input = "What is the answer to the Ultimate Question of Life, the Universe, and Everything? 42"
    val answer = input.substringAfter("?")
    println(answer)
//sampleEnd
}
```

Additionally, you can take a substring after the last occurrence of a character:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "To be, or not to be, that is the question."
    val question = input.substringAfterLast(",")
    println(question)
//sampleEnd
}
```

Use multiline strings

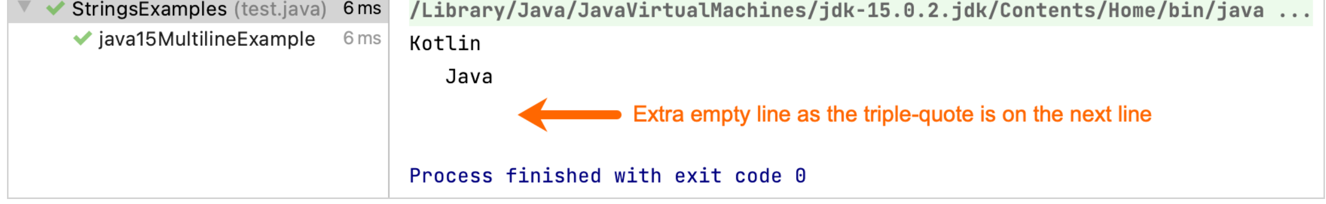
Before Java 15, there were several ways to create a multiline string. For example, using the join() function of the String class:

```
// Java
String lineSeparator = System.getProperty("line.separator");
String result = String.join(lineSeparator,
       "Kotlin",
       "Java");
System.out.println(result);
```

In Java 15, text blocks appeared. There is one thing to keep in mind: if you print a multiline string and the triple-quote is on the next line, there will be an extra empty line:

```
// Java
String result = """
    Kotlin
       Java
    """;
System.out.println(result);
```

The output:



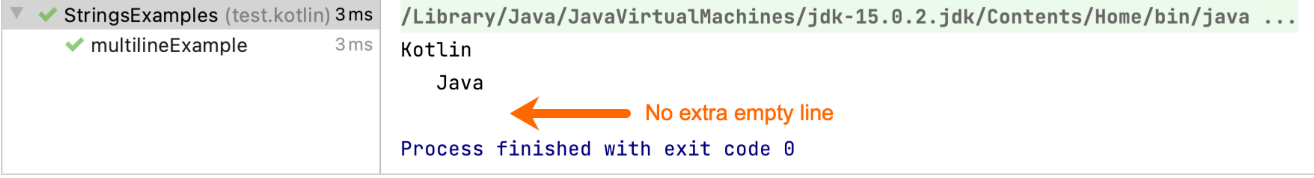
Java 15 multiline output

If you put the triple-quote on the same line as the last word, this difference in behavior disappears.

In Kotlin, you can format your line with the quotes on the new line, and there will be no extra empty line in the output. The left-most character of any line identifies the beginning of the line. The difference with Java is that Java automatically trims indents, and in Kotlin you should do it explicitly:

```
fun main() {
//sampleStart
    // Kotlin
    val result = """
        Kotlin
           Java
    """.trimIndent()
    println(result)
//sampleEnd
}
```

The output:



Kotlin multiline output

To have an extra empty line, you should add this empty line to your multiline string explicitly.

In Kotlin, you can also use the trimMargin() function to customize the indents:

```
// Kotlin
fun main() {
    val result = """
       #  Kotlin
       #  Java
   """.trimMargin("#")
    println(result)
}
```

Learn more about multiline strings.

What's next?

- Look through other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java to Kotlin converter.

If you have a favorite idiom, we invite you to share it by sending a pull request.

Collections in Java and Kotlin

Collections are groups of a variable number of items (possibly zero) that are significant to the problem being solved and are commonly operated on. This guide explains and compares collection concepts and operations in Java and Kotlin. It will help you migrate from Java to Kotlin and write your code in the authentically Kotlin way.

The first part of this guide contains a quick glossary of operations on the same collections in Java and Kotlin. It is divided into operations that are the same and operations that exist only in Kotlin. The second part of the guide, starting from Mutability, explains some of the differences by looking at specific cases.

For an introduction to collections, see the Collections overview or watch this video by Sebastian Aigner, Kotlin Developer Advocate.

> All of the examples below use Java and Kotlin standard library APIs only.

Operations that are the same in Java and Kotlin

In Kotlin, there are many operations on collections that look exactly the same as their counterparts in Java.

Operations on lists, sets, queues, and deques

Description	Common operations	More Kotlin alternatives

Description	Common operations	More Kotlin alternatives
Add an element or elements	add(), addAll()	Use the plusAssign(+=) operator: collection += element, collection += anotherCollection.
Check whether a collection contains an element or elements	contains(), containsAll()	Use the in keyword to call contains() in the operator form: element in collection.
Check whether a collection is empty	isEmpty()	Use isNotEmpty() to check whether a collection is not empty.
Remove under a certain condition	removeIf()	
Leave only selected elements	retainAll()	
Remove all elements from a collection	clear()	
Get a stream from a collection	stream()	Kotlin has its own way to process streams: sequences and methods like map() and filter().
Get an iterator from a collection	iterator()	

Operations on maps

Description	Common operations	More Kotlin alternatives
Add an element or elements	put(), putAll(), putIfAbsent()	In Kotlin, the assignment map[key] = value behaves the same as put(key, value). Also, you may use the plusAssign(+=) operator: map += Pair(key, value) or map += anotherMap.
Replace an element or elements	put(), replace(), replaceAll()	Use the indexing operator map[key] = value instead of put() and replace().
Get an element	get()	Use the indexing operator to get an element: map[index].
Check whether a map contains an element or elements	containsKey(), containsValue()	Use the in keyword to call contains() in the operator form: element in map.
Check whether a map is empty	isEmpty()	Use isNotEmpty() to check whether a map is not empty.
Remove an element	remove(key), remove(key, value)	Use the minusAssign(-=) operator: map -= key.

Description	Common operations	More Kotlin alternatives
Remove all elements from a map	clear()	
Get a stream from a map	stream() on entries, keys, or values	

Operations that exist only for lists

Description	Common operations	More Kotlin alternatives
Get an index of an element	indexOf()	
Get the last index of an element	lastIndexOf()	
Get an element	get()	Use the indexing operator to get an element: list[index].
Take a sublist	subList()	
Replace an element or elements	set(), replaceAll()	Use the indexing operator instead of set(): list[index] = value.

Operations that differ a bit

Operations on any collection type

Description	Java	Kotlin
Get a collection's size	size()	count(), size
Get flat access to nested collection elements	collectionOfCollections.forEach(flatCollection::addAll) or collectionOfCollections.stream().flatMap().collect()	flatten() or flatMap()
Apply the given function to every element	stream().map().collect()	map()
Apply the provided operation to collection elements sequentially and return the accumulated result	stream().reduce()	reduce(), fold()
Group elements by a classifier and count them	stream().collect(Collectors.groupingBy(classifier, counting()))	eachCount()

Description	Java	Kotlin
Filter by a condition	stream().filter().collect()	filter()
Check whether collection elements satisfy a condition	stream().noneMatch(), stream().anyMatch(), stream().allMatch()	none(), any(), all()
Sort elements	stream().sorted().collect()	sorted()
Take the first N elements	stream().limit(N).collect()	take(N)
Take elements with a predicate	stream().takeWhile().collect()	takeWhile()
Skip the first N elements	stream().skip(N).collect()	drop(N)
Skip elements with a predicate	stream().dropWhile().collect()	dropWhile()
Build maps from collection elements and certain values associated with them	stream().collect(toMap(keyMapper, valueMapper))	associate()

To perform all of the operations listed above on maps, you first need to get an entrySet of a map.

Operations on lists

Description	Java	Kotlin
Sort a list into natural order	sort(null)	sort()
Sort a list into descending order	sort(comparator)	sortDescending()
Remove an element from a list	remove(index), remove(element)	removeAt(index), remove(element) or collection -= element
Fill all elements of a list with a certain value	Collections.fill()	fill()
Get unique elements from a list	stream().distinct().toList()	distinct()

Operations that don't exist in Java's standard library

- zip(), unzip() – transform a collection.
- aggregate() – group by a condition.
- takeLast(), takeLastWhile(), dropLast(), dropLastWhile() – take or drop elements by a predicate.

- slice(), chunked(), windowed() – retrieve collection parts.
- Plus (+) and minus (-) operators – add or remove elements.

If you want to take a deep dive into zip(), chunked(), windowed(), and some other operations, watch this video by Sebastian Aigner about advanced collection operations in Kotlin:

Watch video online.

Mutability

In Java, there are mutable collections:

```
// Java
// This list is mutable!
public List<Customer> getCustomers() { ... }
```

Partially mutable ones:

```
// Java
List<String> numbers = Arrays.asList("one", "two", "three", "four");
numbers.add("five"); // Fails in runtime with `UnsupportedOperationException`
```

And immutable ones:

```
// Java
List<String> numbers = new LinkedList<>();
// This list is immutable!
List<String> immutableCollection = Collections.unmodifiableList(numbers);
immutableCollection.add("five"); // Fails in runtime with `UnsupportedOperationException`
```

If you write the last two pieces of code in IntelliJ IDEA, the IDE will warn you that you're trying to modify an immutable object. This code will compile and fail in runtime with UnsupportedOperationException. You can't tell whether a collection is mutable by looking at its type.

Unlike in Java, in Kotlin you explicitly declare mutable or read-only collections depending on your needs. If you try to modify a read-only collection, the code won't compile:

```
// Kotlin
val numbers = mutableListOf("one", "two", "three", "four")
numbers.add("five")            // This is OK
val immutableNumbers = listOf("one", "two")
//immutableNumbers.add("five") // Compilation error - Unresolved reference: add
```

Read more about immutability on the Kotlin coding conventions page.

Covariance

In Java, you can't pass a collection with a descendant type to a function that takes a collection of the ancestor type. For example, if Rectangle extends Shape, you can't pass a collection of Rectangle elements to a function that takes a collection of Shape elements. To make the code compilable, use the ? extends Shape type so the function can take collections with any inheritors of Shape:

```
// Java
class Shape {}

class Rectangle extends Shape {}

public void doSthWithShapes(List<? extends Shape> shapes) {
/* If using just List<Shape>, the code won't compile when calling
this function with the List<Rectangle> as the argument as below */
}

public void main() {
    var rectangles = List.of(new Rectangle(), new Rectangle());
    doSthWithShapes(rectangles);
}
```

In Kotlin, read-only collection types are covariant. This means that if a Rectangle class inherits from the Shape class, you can use the type List<Rectangle> anywhere the List<Shape> type is required. In other words, the collection types have the same subtyping relationship as the element types. Maps are covariant on the value type, but not on the key type. Mutable collections aren't covariant – this would lead to runtime failures.

```
// Kotlin
open class Shape(val name: String)

class Rectangle(private val rectangleName: String) : Shape(rectangleName)

fun doSthWithShapes(shapes: List<Shape>) {
    println("The shapes are: ${shapes.joinToString { it.name }}")
}

fun main() {
    val rectangles = listOf(Rectangle("rhombus"), Rectangle("parallelepiped"))
    doSthWithShapes(rectangles)
}
```

Read more about collection types here.

Ranges and progressions

In Kotlin, you can create intervals using ranges. For example, Version(1, 11)..Version(1, 30) includes all of the versions from 1.11 to 1.30. You can check that your version is in the range by using the in operator: Version(0, 9) in versionRange.

In Java, you need to manually check whether a Version fits both bounds:

```
// Java
class Version implements Comparable<Version> {

    int major;
    int minor;

    Version(int major, int minor) {
        this.major = major;
        this.minor = minor;
    }

    @Override
    public int compareTo(Version o) {
        if (this.major != o.major) {
            return this.major - o.major;
        }
        return this.minor - o.minor;
    }
}

public void compareVersions() {
    var minVersion = new Version(1, 11);
    var maxVersion = new Version(1, 31);
```

```
    System.out.println(
            versionIsInRange(new Version(0, 9), minVersion, maxVersion));
    System.out.println(
            versionIsInRange(new Version(1, 20), minVersion, maxVersion));
}

public Boolean versionIsInRange(Version versionToCheck, Version minVersion,
                                Version maxVersion) {
    return versionToCheck.compareTo(minVersion) >= 0
            && versionToCheck.compareTo(maxVersion) <= 0;
}
```

In Kotlin, you operate with a range as a whole object. You don't need to create two variables and compare a Version with them:

```
// Kotlin
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        }
        return this.minor - other.minor
    }
}

fun main() {
    val versionRange = Version(1, 11)..Version(1, 30)

    println(Version(0, 9) in versionRange)
    println(Version(1, 20) in versionRange)
}
```

As soon as you need to exclude one of the bounds, like to check whether a version is greater than or equal to (>=) the minimum version and less than (<) the maximum version, these inclusive ranges won't help.

Comparison by several criteria

In Java, to compare objects by several criteria, you may use the comparing() and thenComparingX() functions from the Comparator interface. For example, to compare people by their name and age:

```
class Person implements Comparable<Person> {
    String name;
    int age;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return this.name + " " + age;
    }
}

public void comparePersons() {
    var persons = List.of(new Person("Jack", 35), new Person("David", 30),
            new Person("Jack", 25));
    System.out.println(persons.stream().sorted(Comparator
            .comparing(Person::getName)
            .thenComparingInt(Person::getAge)).collect(toList()));
}
```

In Kotlin, you just enumerate which fields you want to compare:

```
data class Person(
```

```
    val name: String,
    val age: Int
)

fun main() {
    val persons = listOf(Person("Jack", 35), Person("David", 30),
        Person("Jack", 25))
    println(persons.sortedWith(compareBy(Person::name, Person::age)))
}
```

Sequences

In Java, you can generate a sequence of numbers this way:

```
// Java
int sum = IntStream.iterate(1, e -> e + 3)
    .limit(10).sum();
System.out.println(sum); // Prints 145
```

In Kotlin, use sequences. Multi-step processing of sequences is executed lazily when possible – actual computing happens only when the result of the whole processing chain is requested.

```
fun main() {
//sampleStart
    // Kotlin
    val sum = generateSequence(1) {
        it + 3
    }.take(10).sum()
    println(sum) // Prints 145
//sampleEnd
}
```

Sequences may reduce the number of steps that are needed to perform some filtering operations. See the sequence processing example, which shows the difference between Iterable and Sequence.

Removal of elements from a list

In Java, the remove() function accepts an index of an element to remove.

When removing an integer element, use the Integer.valueOf() function as the argument for the remove() function:

```
// Java
public void remove() {
    var numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    numbers.add(1);
    numbers.remove(1); // This removes by index
    System.out.println(numbers); // [1, 3, 1]
    numbers.remove(Integer.valueOf(1));
    System.out.println(numbers); // [3, 1]
}
```

In Kotlin, there are two types of element removal: by index with removeAt() and by value with remove().

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = mutableListOf(1, 2, 3, 1)
    numbers.removeAt(0)
    println(numbers) // [2, 3, 1]
    numbers.remove(1)
    println(numbers) // [2, 3]
//sampleEnd
}
```

Traverse a map

In Java, you can traverse a map via forEach:

```
// Java
numbers.forEach((k,v) -> System.out.println("Key = " + k + ", Value = " + v));
```

In Kotlin, use a for loop or a forEach, similar to Java's forEach, to traverse a map:

```
// Kotlin
for ((k, v) in numbers) {
    println("Key = $k, Value = $v")
}
// Or
numbers.forEach { (k, v) -> println("Key = $k, Value = $v") }
```

Get the first and the last items of a possibly empty collection

In Java, you can safely get the first and the last items by checking the size of the collection and using indices:

```
// Java
var list = new ArrayList<>();
//...
if (list.size() > 0) {
    System.out.println(list.get(0));
    System.out.println(list.get(list.size() - 1));
}
```

You can also use the getFirst() and getLast() functions for Deque and its inheritors:

```
// Java
var deque = new ArrayDeque<>();
//...
if (deque.size() > 0) {
    System.out.println(deque.getFirst());
    System.out.println(deque.getLast());
}
```

In Kotlin, there are the special functions firstOrNull() and lastOrNull(). Using the Elvis operator, you can perform further actions right away depending on the result of a function. For example, firstOrNull():

```
// Kotlin
val emails = listOf<String>() // Might be empty
val theOldestEmail = emails.firstOrNull() ?: ""
val theFreshestEmail = emails.lastOrNull() ?: ""
```

Create a set from a list

In Java, to create a Set from a List, you can use the Set.copyOf function:

```
// Java
public void listToSet() {
    var sourceList = List.of(1, 2, 3, 1);
    var copySet = Set.copyOf(sourceList);
    System.out.println(copySet);
}
```

In Kotlin, use the function toSet():

```
fun main() {
//sampleStart
    // Kotlin
    val sourceList = listOf(1, 2, 3, 1)
    val copySet = sourceList.toSet()
```

```
    println(copySet)
//sampleEnd
}
```

Group elements

In Java, you can group elements with the Collectors function groupingBy():

```
// Java
public void analyzeLogs() {
    var requests = List.of(
        new Request("https://kotlinlang.org/docs/home.html", 200),
        new Request("https://kotlinlang.org/docs/home.html", 400),
        new Request("https://kotlinlang.org/docs/comparison-to-java.html", 200)
    );
    var urlsAndRequests = requests.stream().collect(
            Collectors.groupingBy(Request::getUrl));
    System.out.println(urlsAndRequests);
}
```

In Kotlin, use the function groupBy():

```
class Request(
    val url: String,
    val responseCode: Int
)

fun main() {
//sampleStart
    // Kotlin
    val requests = listOf(
        Request("https://kotlinlang.org/docs/home.html", 200),
        Request("https://kotlinlang.org/docs/home.html", 400),
        Request("https://kotlinlang.org/docs/comparison-to-java.html", 200)
    )
    println(requests.groupBy(Request::url))
//sampleEnd
}
```

Filter elements

In Java, to filter elements from a collection, you need to use the Stream API. The Stream API has intermediate and terminal operations. filter() is an intermediate operation, which returns a stream. To receive a collection as the output, you need to use a terminal operation, like collect(). For example, to leave only those pairs whose keys end with 1 and whose values are greater than 10:

```
// Java
public void filterEndsWith() {
    var numbers = Map.of("key1", 1, "key2", 2, "key3", 3, "key11", 11);
    var filteredNumbers = numbers.entrySet().stream()
        .filter(entry -> entry.getKey().endsWith("1") && entry.getValue() > 10)
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
    System.out.println(filteredNumbers);
}
```

In Kotlin, filtering is built into collections, and filter() returns the same collection type that was filtered. So, all you need to write is the filter() and its predicate:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredNumbers = numbers.filter { (key, value) -> key.endsWith("1") && value > 10 }
    println(filteredNumbers)
//sampleEnd
}
```

Learn more about filtering maps here.

Filter elements by type

In Java, to filter elements by type and perform actions on them, you need to check their types with the instanceof operator and then do the type cast:

```
// Java
public void objectIsInstance() {
    var numbers = new ArrayList<>();
    numbers.add(null);
    numbers.add(1);
    numbers.add("two");
    numbers.add(3.0);
    numbers.add("four");
    System.out.println("All String elements in upper case:");
    numbers.stream().filter(it -> it instanceof String)
        .forEach( it -> System.out.println(((String) it).toUpperCase()));
}
```

In Kotlin, you just call filterIsInstance<NEEDED_TYPE>() on your collection, and the type cast is done by Smart casts:

```
// Kotlin
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf(null, 1, "two", 3.0, "four")
    println("All String elements in upper case:")
    numbers.filterIsInstance<String>().forEach {
        println(it.uppercase())
    }
//sampleEnd
}
```

Test predicates

Some tasks require you to check whether all, none, or any elements satisfy a condition. In Java, you can do all of these checks via the Stream API functions allMatch(), noneMatch(), and anyMatch():

```
// Java
public void testPredicates() {
    var numbers = List.of("one", "two", "three", "four");
    System.out.println(numbers.stream().noneMatch(it -> it.endsWith("e"))); // false
    System.out.println(numbers.stream().anyMatch(it -> it.endsWith("e"))); // true
    System.out.println(numbers.stream().allMatch(it -> it.endsWith("e"))); // false
}
```

In Kotlin, the extension functions none(), any(), and all() are available for every Iterable object:

```
fun main() {
//sampleStart
// Kotlin
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.none { it.endsWith("e") })
    println(numbers.any { it.endsWith("e") })
    println(numbers.all { it.endsWith("e") })
//sampleEnd
}
```

Learn more about test predicates.

Collection transformation operations

Zip elements

In Java, you can make pairs from elements with the same positions in two collections by iterating simultaneously over them:

```
// Java
public void zip() {
    var colors = List.of("red", "brown");
    var animals = List.of("fox", "bear", "wolf");
```

```
    for (int i = 0; i < Math.min(colors.size(), animals.size()); i++) {
        String animal = animals.get(i);
        System.out.println("The " + animal.substring(0, 1).toUpperCase()
                + animal.substring(1) + " is " + colors.get(i));
    }
}
```

If you want to do something more complex than just printing pairs of elements into the output, you can use Records. In the example above, the record would be record AnimalDescription(String animal, String color) {}.

In Kotlin, use the zip() function to do the same thing:

```
fun main() {
//sampleStart
    // Kotlin
    val colors = listOf("red", "brown")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal ->
        "The ${animal.replaceFirstChar { it.uppercase() }} is $color" })
//sampleEnd
}
```

zip() returns the List of Pair objects.

If collections have different sizes, the result of zip() is the smaller size. The last elements of the larger collection are not included in the result.

Associate elements

In Java, you can use the Stream API to associate elements with characteristics:

```
// Java
public void associate() {
    var numbers = List.of("one", "two", "three", "four");
    var wordAndLength = numbers.stream()
        .collect(toMap(number -> number, String::length));
    System.out.println(wordAndLength);
}
```

In Kotlin, use the associate() function:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

What's next?

- Visit Kotlin Koans – complete exercises to learn Kotlin syntax. Each exercise is created as a failing unit test and your job is to make it pass.
- Look through other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java to Kotlin converter.
- Discover collections in Kotlin.

If you have a favorite idiom, we invite you to share it by sending a pull request.

Nullability in Java and Kotlin

Nullability is the ability of a variable to hold a null value. When a variable contains null, an attempt to dereference the variable leads to a NullPointerException. There

are many ways to write code in order to minimize the probability of receiving null pointer exceptions.

This guide covers differences between Java's and Kotlin's approaches to handling possibly nullable variables. It will help you migrate from Java to Kotlin and write your code in authentic Kotlin style.

The first part of this guide covers the most important difference – support for nullable types in Kotlin and how Kotlin processes types from Java code. The second part, starting from Checking the result of a function call, examines several specific cases to explain certain differences.

Learn more about null safety in Kotlin.

Support for nullable types

The most important difference between Kotlin's and Java's type systems is Kotlin's explicit support for nullable types. It is a way to indicate which variables can possibly hold a null value. If a variable can be null, it's not safe to call a method on the variable because this can cause a NullPointerException. Kotlin prohibits such calls at compile time and thereby prevents lots of possible exceptions. At runtime, objects of nullable types and objects of non-nullable types are treated the same: A nullable type isn't a wrapper for a non-nullable type. All checks are performed at compile time. That means there's almost no runtime overhead for working with nullable types in Kotlin.

> We say "almost" because, even though intrinsic checks are generated, their overhead is minimal.

In Java, if you don't write null checks, methods may throw a NullPointerException:

```
// Java
int stringLength(String a) {
    return a.length();
}

void main() {
    stringLength(null); // Throws a `NullPointerException`
}
```

This call will have the following output:

```
java.lang.NullPointerException: Cannot invoke "String.length()" because "a" is null
    at test.java.Nullability.stringLength(Nullability.java:8)
    at test.java.Nullability.main(Nullability.java:12)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
```

In Kotlin, all regular types are non-nullable by default unless you explicitly mark them as nullable. If you don't expect a to be null, declare the stringLength() function as follows:

```
// Kotlin
fun stringLength(a: String) = a.length
```

The parameter a has the String type, which in Kotlin means it must always contain a String instance and it cannot contain null. Nullable types in Kotlin are marked with a question mark ?, for example, String?. The situation with a NullPointerException at runtime is impossible if a is String because the compiler enforces the rule that all arguments of stringLength() not be null.

An attempt to pass a null value to the stringLength(a: String) function will result in a compile-time error, "Null can not be a value of a non-null type String":

```
fun main() {
    stringLength( a: null)
}
}
```

Null can not be a value of a non-null type String

Change parameter 'a' type of function 'stringLength' to 'String?' ⌥⇧↩ More actions... ⌥↩

```
public final fun stringLength(
    a: String
): Int
```

java-to-kotlin-idioms-guide

Passing null to a non-nullable function error

If you want to use this function with any arguments, including null, use a question mark after the argument type String? and check inside the function body to ensure that the value of the argument is not null:

```
// Kotlin
fun stringLength(a: String?): Int = if (a != null) a.length else 0
```

After the check is passed successfully, the compiler treats the variable as if it were of the non-nullable type String in the scope where the compiler performs the check.

If you don't perform this check, the code will fail to compile with the following message: "Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?".

You can write the same shorter – use the safe-call operator ?. (If-not-null shorthand), which allows you to combine a null check and a method call into a single operation:

```
// Kotlin
fun stringLength(a: String?): Int = a?.length ?: 0
```

Platform types

In Java, you can use annotations showing whether a variable can or cannot be null. Such annotations aren't part of the standard library, but you can add them separately. For example, you can use the JetBrains annotations @Nullable and @NotNull (from the org.jetbrains.annotations package) or annotations from Eclipse (org.eclipse.jdt.annotation). Kotlin can recognize such annotations when you're calling Java code from Kotlin code and will treat types according to their annotations.

If your Java code doesn't have these annotations, then Kotlin will treat Java types as platform types. But since Kotlin doesn't have nullability information for such types, its compiler will allow all operations on them. You will need to decide whether to perform null checks, because:

- Just as in Java, you'll get a NullPointerException if you try to perform an operation on null.
- The compiler won't highlight any redundant null checks, which it normally does when you perform a null-safe operation on a value of a non-nullable type.

Learn more about calling Java from Kotlin in regard to null-safety and platform types.

Checking the result of a function call

One of the most common situations where you need to check for null is when you obtain a result from a function call.

In the following example, there are two classes, Order and Customer. Order has a reference to an instance of Customer. The findOrder() function returns an instance of the Order class, or null if it can't find the order. The objective is to process the customer instance of the retrieved order.

Here are the classes in Java:

```
//Java
record Order (Customer customer) {}

record Customer (String name) {}
```

In Java, call the function and do an if-not-null check on the result to proceed with the dereferencing of the required property:

```
// Java
Order order = findOrder();

if (order != null) {
    processCustomer(order.getCustomer());
}
```

Converting the Java code above to Kotlin code directly results in the following:

```
// Kotlin
data class Order(val customer: Customer)

data class Customer(val name: String)

val order = findOrder()

// Direct conversion
if (order != null){
    processCustomer(order.customer)
}
```

Use the safe-call operator ?. (If-not-null shorthand) in combination with any of the scope functions from the standard library. The let function is usually used for this:

```
// Kotlin
val order = findOrder()

order?.let {
    processCustomer(it.customer)
}
```

Here is a shorter version of the same:

```
// Kotlin
findOrder()?.customer?.let(::processCustomer)
```

Default values instead of null

Checking for null is often used in combination with setting the default value in case the null check is successful.

The Java code with a null check:

```
// Java
Order order = findOrder();
if (order == null) {
    order = new Order(new Customer("Antonio"))
}
```

To express the same in Kotlin, use the Elvis operator (If-not-null-else shorthand):

```
// Kotlin
val order = findOrder() ?: Order(Customer("Antonio"))
```

Functions returning a value or null

In Java, you need to be careful when working with list elements. You should always check whether an element exists at an index before you attempt to use the element:

```
// Java
var numbers = new ArrayList<Integer>();
numbers.add(1);
numbers.add(2);
```

```
System.out.println(numbers.get(0));
//numbers.get(5) // Exception!
```

The Kotlin standard library often provides functions whose names indicate whether they can possibly return a null value. This is especially common in the collections API:

```
fun main() {
//sampleStart
    // Kotlin
    // The same code as in Java:
    val numbers = listOf(1, 2)

    println(numbers[0])  // Can throw IndexOutOfBoundsException if the collection is empty
    //numbers.get(5)     // Exception!

    // More abilities:
    println(numbers.firstOrNull())
    println(numbers.getOrNull(5)) // null
//sampleEnd
}
```

Aggregate operations

When you need to get the biggest element or null if there are no elements, in Java you would use the Stream API:

```
// Java
var numbers = new ArrayList<Integer>();
var max = numbers.stream().max(Comparator.naturalOrder()).orElse(null);
System.out.println("Max: " + max);
```

In Kotlin, use aggregate operations:

```
// Kotlin
val numbers = listOf<Int>()
println("Max: ${numbers.maxOrNull()}")
```

Learn more about collections in Java and Kotlin.

Casting types safely

When you need to safely cast a type, in Java you would use the instanceof operator and then check how well it worked:

```
// Java
int getStringLength(Object y) {
    return y instanceof String x ? x.length() : -1;
}

void main() {
    System.out.println(getStringLength(1)); // Prints `-1`
}
```

To avoid exceptions in Kotlin, use the safe cast operator as?, which returns null on failure:

```
// Kotlin
fun main() {
    println(getStringLength(1)) // Prints `-1`
}

fun getStringLength(y: Any): Int {
    val x: String? = y as? String // null
    return x?.length ?: -1 // Returns -1 because `x` is null
}
```

In the Java example above, the function getStringLength() returns a result of the primitive type int. To make it return null, you can use the boxed type Integer. However, it's more resource-efficient to make such functions return a negative value and then check the value – you would do the check anyway, but no additional boxing is performed this way.

What's next?

- Browse other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java-to-Kotlin (J2K) converter.
- Check out other migration guides:
  - Strings in Java and Kotlin
  - Collections in Java and Kotlin

If you have a favorite idiom, feel free to share it with us by sending a pull request!

Introduction

A good library is one that has:

- Backward compatibility
- Complete and easy-to-understand documentation
- Minimum cognitive complexity
- Consistent API

This guide contains a summary of best practices and ideas to consider when writing an API for your library. It consists of the following chapters:

- Readability
- Predictability
- Debuggability
- Backward compatibility

Many of the following best practices provide advice on how to reduce the cognitive complexity of an API. As such, this guide provides an explanation of cognitive complexity before proceeding to best practices.

Cognitive complexity

Cognitive complexity is the amount of mental effort a person needs to spend to understand a piece of code. A codebase with high cognitive complexity is more difficult to understand and maintain, which can lead to bugs and delays in development.

An example of high cognitive complexity is a class or module that does not follow the Single Responsibility Principle. A class or module that does too many things is hard to understand and modify. In contrast, a class or module that has one clear and well-defined responsibility is easier to work with.

Functions can also have high cognitive complexity. Some traits of a "badly written" function are:

- Too many arguments, variables, or loops.
- Complex logic with many nested if-else statements.

A function like that is harder to work with than a function with clear and simple logic – one with few parameters and an easy-to-understand control flow. An example of high cognitive complexity:

```
fun processData(
    data: List<String>,
    delimiter: String,
    ignoreCase: Boolean,
```

```
    sort: Boolean,
    maxLength: Int
) {
    // Some complex processing logic
}
```

Decomposing this functionality lowers the cognitive complexity:

```
fun delimit(data: List<String>, delimiter: String) { … }
fun ignoreCase(data: List<String>) { … }
fun sortAscending(data: List<String>) { … }
fun sortDescending(data: List<String>) { … }
fun maxLength(data: List<String>, maxLength: Int) { … }
```

You can simplify the code above even more with the help of extension functions:

```
fun List<String>.delimit(delimiter: String): List<String> { … }
fun List<String>.sortAscending(): List<String> { … }
fun List<String>.sortDescending(): List<String> { … }
fun List<String>.maxLength(maxLength: Int): List<String> { … }
…
```

What's next?

Learn about APIs' readability.

Readability

This chapter discusses considerations about API consistency and provides the following recommendations:

- Use a builder DSL
- Use constructor-like functions where applicable
- Use member and extension functions appropriately
- Avoid using Boolean arguments in functions

API consistency

A consistent and well-documented API is crucial for a good development experience. The same is valid for argument order, overall naming scheme, and overloads. Also, it's worth documenting all conventions.

For example, if one of your methods accepts offset and length as parameters, then so should other methods, instead of, for example, accepting startIndex and endIndex. Parameters like these are most likely of Int or Long type, and thus it's very easy to confuse them.

The same works for parameter order: Keep it consistent between methods and overloads. Otherwise, users of your library might incorrectly guess the order they should pass arguments in.

Here is an example that preserves the parameter order and uses consistent naming:

```
fun String.chop(length: Int): String = substring(0, length)
fun String.chop(length: Int, startIndex: Int) =
    substring(startIndex, length + startIndex)
```

If you have many lookalike methods, name them consistently and predictably. This is how the stdlib API works: There are methods first() and firstOrNull(), single() and singleOrNull(), and so on. It's clear from their names that they are all pairs, and some of them might return null while others might throw an exception.

Use a builder DSL

"Builder" is a well-known pattern in development. It allows you to build a complex entity not in a single expression, but gradually while getting more information.

When you need to use a builder, it's better to write it using a builder DSL, which is binary-compatible and more idiomatic.

A canonical example of a Kotlin builder DSL is kotlinx.html. Consider the following example:

```
header("modal-card-head") {
    p("modal-card-title") {
        +book.book.name
    }
    button(classes = "delete") {
        attributes["aria-label"] = "close"
        attributes["_"] = closeModalScript
    }
}
```

It could be implemented as a traditional builder, but that would be considerably more verbose:

```
headerBuilder()
    .addClasses("modal-card-head")
    .addElement(
        pBuilder()
            .addClasses("modal-card-title")
            .addContent(book.book.name)
            .build()
    )
    .addElement(
        buttonBuilder()
            .addClasses("delete")
            .addAttribute("aria-label", "close")
            .addAttribute("_", closeModalScript)
            .build()
    )
    .build()
```

This implementation has too many details that you don't necessarily need to know, and it requires you to build each entity at the end.

The situation gets even worse if you need to generate a builder's content dynamically in a loop. In this scenario, you have to instantiate a variable and dynamically overwrite it:

```
var buttonBuilder = buttonBuilder()
    .addClasses("delete")
for ((attributeName, attributeValue) in attributes) {
    buttonBuilder = buttonBuilder.addAttribute(attributeName, attributeValue)
}
buttonBuilder.build()
```

Inside the builder DSL, you can directly call a loop and all necessary DSL calls:

```
div("tags") {
    for (genre in book.genres) {
        span("tag is-rounded is-normal is-info is-light") {
            +genre
        }
    }
}
```

Keep in mind that inside curly braces it's impossible to check at compile time whether you have set all the required attributes. To avoid this, pass required arguments as function arguments, not as builder's properties. For example, if you want href to be a mandatory HTML attribute, your function will look like:

```
fun a(href: String, block: A.() -> Unit): A
```

And not just:

```
fun a(block: A.() -> Unit): A
```

> Builder DSLs are backward-compatible as long as you don't delete anything from them. Typically this isn't a problem, because most developers only add more properties to their builder classes over time.

Use constructor-like functions where applicable

Sometimes, you can simplify your API's appearance by using constructor-like functions. A constructor-like function is a function whose name starts with a capital letter, so it looks like a constructor. This approach can make your library easier to understand.

Suppose you want to introduce an Option type in your library:

```
sealed interface Option<T>
class Some<T : Any>(val t: T) : Option<T>
object None : Option<Nothing>
```

You can define implementations of all the Option interface methods – map(), flatMap(), and so on. However, each time your API users create such an Option, they must write extra logic to check what they create. For example:

```
fun findById(id: Int): Option<Person> {
    val person = db.personById(id)
    return if (person == null) None else Some(person)
}
```

To save your users from having to write the same check each time, you can add just one line to your API:

```
fun <T> Option(t: T?): Option<out T & Any> =
    if (t == null) None else Some(t)

// Usage of the code above:
fun findById(id: Int): Option<Person> = Option(db.personById(id))
```

Now, creating a valid Option is as simple as can be: Just call Option(x) and you have a null-safe, purely functional Option idiom.

Another use case for using a constructor-like function is when you need to return "hidden" things, such as a private instance or an internal object. For example, let's look at a method from the standard library:

```
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.isNotEmpty()) elements.asList() else emptyList()
```

In the code above, emptyList() returns the following:

```
internal object EmptyList : List<Nothing>, Serializable, RandomAccess
```

You can write a constructor-like function to lower the cognitive complexity of your code and reduce the size of your API:

```
fun <T> List(): List<T> = EmptyList

// Usage of the code above:
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.isNotEmpty()) elements.asList() else List()
```

Use member and extension functions appropriately

Write only the very core of the API as member functions, and everything else as extension functions. This will help you clearly show to the reader what is the core functionality and what isn't.

For example, consider the following class for a graph:

```
class Graph {
    private val _vertices: MutableSet<Int> = mutableSetOf()
    private val _edges: MutableMap<Int, MutableSet<Int>> = mutableMapOf()

    fun addVertex(vertex: Int) {
        _vertices.add(vertex)
    }

    fun addEdge(vertex1: Int, vertex2: Int) {
        _vertices.add(vertex1)
        _vertices.add(vertex2)
```

```kotlin
        _edges.getOrPut(vertex1) { mutableSetOf() }.add(vertex2)
        _edges.getOrPut(vertex2) { mutableSetOf() }.add(vertex1)
    }

    val vertices: Set<Int> get() = _vertices
    val edges: Map<Int, Set<Int>> get() = _edges
}
```

This class contains a bare minimum of vertices and edges as private variables, functions to add vertices and edges, and accessor functions that return an immutable representation of the current state.

You can add all the remaining functionality outside the class:

```kotlin
fun Graph.getNumberOfVertices(): Int = vertices.size
fun Graph.getNumberOfEdges(): Int = edges.size
fun Graph.getDegree(vertex: Int): Int = edges[vertex]?.size ?: 0
```

Only properties, overrides, and accessors should be members.

Avoid using Boolean arguments in functions

Ideally, a reader should be able to tell the purpose of a function argument just by reading code. With Boolean arguments, however, this is almost impossible to do, especially if you're not using an IDE (for example, if you're reviewing the code in a version control service). Using named arguments can help clarify the purpose of arguments, but for now there is no way to force developers to use them in IDEs. Another option is to create a function that contains the action of the Boolean argument and give this function a descriptive name.

For example, in the standard library there are two functions for map():

```kotlin
fun map(transform: (T) -> R): List<R>

fun mapNotNull(transform: (T) -> R?): List<R>
```

It was possible to add something like map(filterNulls: Boolean) and write code like this:

```kotlin
listOf(1, null, 2).map(false) { it.toString() }
```

From reading this code, it's difficult to infer what false refers to. However, if you use the mapNotNull() function, readers will be able to understand the logic at a glance:

```kotlin
listOf(1, null, 2).mapNotNull { it.toString() }
```

What's next?

Learn about APIs':

- Predictability
- Debuggability
- Backward compatibility

Predictability

This chapter contains the following recommendations:

- Use sealed interfaces
- Hide implementations with sealed classes
- Validate your inputs and state

- Validate inputs with the require() function
- Validate state with the check() function
- Avoid arrays in public signatures
- Avoid varargs

Use sealed interfaces

Interfaces in your API are usually necessary when you need to have an abstraction from an implementation. If you have to use interfaces, consider using sealed interfaces. This is especially important if you don't want your API's users to extend your hierarchy.

> Remember that adding a new implementation to a sealed interface will immediately make a user's existing code invalid.

For example, JSON types can be of six types: object, array, number, string, boolean, and null. Creating a generic interface JsonElement can result in errors because a user can accidentally define a new implementation of JsonElement, which could break your code. Instead, you can make interface JsonElement sealed and add an implementation for each type:

```
sealed interface JsonElement

class JsonNumber(val value: Number) : JsonElement
class JsonObject(val values: Map<String, JsonElement>) : JsonElement
class JsonArray(val values: List<JsonElement>) : JsonElement
class JsonBoolean(val value: Boolean) : JsonElement
class JsonString(val value: String) : JsonElement
object JsonNull : JsonElement
```

This approach helps you avoid mistakes on both the library and the client sides.

The key benefit of using sealed types comes into play when you use them in a when expression. If it's possible to verify that the statement covers all cases, you don't need to add an else clause to the statement:

```
fun processJson(json: JsonElement) = when (json) {
    is JsonNumber -> { /* Process as a number */ }
    is JsonObject -> { /* Process as an object */ }
    is JsonArray -> { /* Process as an array */ }
    is JsonBoolean -> { /* Process as a boolean */ }
    is JsonString -> { /* Process as a string */ }
    is JsonNull -> { /* Process as null */ }
    // `else` clause is not required because all the cases are covered
}
```

Hide implementations with sealed classes

If you have a sealed interface in your API, it doesn't mean that you should expose all its implementations in your API, too. Minimizing is typically better. If you need to avoid leaky abstractions or want to prevent API users from extending your interfaces, consider using sealed classes or interfaces with your internal implementations, too.

For example, a library that works with different databases can have an interface of a database response like this:

```
sealed interface DBResponse {
    operator fun <T> get(columnName: String): Sequence<T>
}
```

Exposing implementations of this interface, such as SQLiteResponse or MongoResponse, to API users is a leaky abstraction, and it complicates the support of this API. In such a library, you might handle only your implementations of DBResponse. If a user passes their implementation of DBResponse into a library's method accepting responses, it can cause an error. Using sealed interfaces and classes prevents this.

Validate your inputs and state

Validate inputs with the require() function

It's possible to misuse an API. To help your users work with your API correctly, you should validate inputs as early as possible with the require() function.

For example, this is a simple library function that saves users to some external API:

```
fun saveUser(username: String, password: String) {
    api.saveUser(User(username, password))
}
```

You should perform validation on the function's arguments to make sure that everything behaves as expected. For example, check that username is unique and not empty, even if you have already defined these constraints in your database:

```
fun saveUser(username: String, password: String) {
    require(username.isNotBlank()) { "Username should not be blank" }
    require(api.usernameAvailable(username)) { "Username $username is already taken" }
    require(password.isNotBlank()) { "Password should not be blank" }
    require(password.length > 6) { "Password should contain at least 7 letters" }
    require(
        /* Some complex check */
    ) { "..." }

    api.saveUser(User(username, password))
}
```

This way you ensure that your user doesn't need to dig into complex stack traces that lead to the database. In the event of an exception, it will be an IllegalArgumentException with a meaningful message, not a generic database exception.

> If you have implemented input validation, you should also document these checks.

Validate state with the check() function

The same recommendations apply to checking the internal state. The most obvious example is InputStream because you can't read from a closed input stream.

Consider the class InputStream with a readByte() method and its usage:

```
class InputStream : Closeable {
    private var open = true
    fun readByte(): Byte { /* Read and return one byte */ }
    override fun close() {
        // Dispose of the underlying resource
        open = false
    }
}

fun readTwoBytes(inputStream: InputStream): Pair<Byte, Byte> {
    val first = inputStream.use { it.readByte() }
    val second = inputStream.readByte()
    return Pair(first, second)
}
```

The readTwoBytes() method has to throw an IllegalStateException because use{} closes a Closeable input stream, and a user shouldn't be able to read from a closed stream. To implement this, modify the code of the readByte() function:

```
fun readByte(): Byte {
    check(open) { "Can't read from the already closed stream" }
    // Read and return one byte
}
```

In the example above, the check() function is used, not require(). These functions throw different exceptions: require() throws an IllegalArgumentException, whereas check() throws an IllegalStateException. This difference might become significant when debugging.

Avoid arrays in public signatures

Arrays are always mutable, and Kotlin is built around safe – read-only or immutable – objects. If you have to use arrays in your API, copy them before passing them anywhere so that you can check that they have not been modified. As an alternative, use read-only and mutable collections according to your intentions. Generally,

it is best to avoid using arrays, and if you must, do so with extra caution.

For example, enum classes in Kotlin have the values() function that returns an array of all elements of the enum. If the array is not copied, a user is able to rewrite the elements:

```
enum class Test { A, B }

fun main() { Test.values()[0] = Test.B }
```

If you cache values inside the enum, the cache will be corrupted after running the code above. If the values are not cached, it's an additional runtime overhead for each call of the values() function.

Avoid varargs

A vararg – variable number of arguments – works as an array under the hood, but the array elements are passed individually to the function, not the whole array. This operation is costly because it's copying the same array repeatedly.

Consider the following code:

```
fun printElements(delimiter: String, vararg elements: String) {
    for (i in elements.indices) {
        print(elements[i])
        if (i < elements.lastIndex) print(delimiter)
    }
}

fun printWithSpace(vararg elements: String) {
    printElements(" ", *elements)
}

fun main() {
    printWithSpace("x", "y", "z")
}
```

The printElements() function prints all strings from the vararg argument elements with a delimiter, and the printWithSpace() function calls printElements() with the delimiter defined as a space. The code looks innocent: you just pass elements from printWithSpace() to printElements(). Without the spread operator *, the code won't compile, but with it, the array is actually copied before being passed to the printElements() function. The longer the chain is, the more copies are created and the bigger the unexpected memory overhead is.

What's next?

Learn about APIs':

- Debuggability
- Backward compatibility

Debuggability

This chapter contains considerations about debuggability.

Always provide a toString() method

To make debugging easier, add a toString() implementation to every class you introduce, even to internal ones. If toString() is part of a contract, document it explicitly.

The following code is a simplified example from a graphical modeling area:

```
class Vector2D(val x: Int, val y: Int)

fun main() {
    val result = (1..20).map { Vector2D(it, it) }
```

```
    println(result)
}
```

The output of this code is not very useful:

```
[Vector2D@27bc2616, Vector2D@3941a79c, Vector2D@506e1b77,...]
```

Neither is the information provided in the debug tool window:

```
0 = {Vector2D@836} Vector2D@b4c966a
1 = {Vector2D@837} Vector2D@2f4d3709
2 = {Vector2D@838} Vector2D@4e50df2e
3 = {Vector2D@839} Vector2D@1d81eb93
4 = {Vector2D@840} Vector2D@7291c18f
5 = {Vector2D@841} Vector2D@34a245ab
```

Vector class objects in the debug tool window

To make both logging and debugging much more readable, add a simple toString() implementation like this:

```
override fun toString(): String =
    "Vector2D(x=$x, y=$y)"
```

This results in improved output:

```
[Vector2D(x=1, y=1), Vector2D(x=2, y=2), Vector2D(x=3, y=3), ...
```

```
0 = {Vector2D@836} Vector2D(x=1, y=1)
1 = {Vector2D@837} Vector2D(x=2, y=2)
2 = {Vector2D@838} Vector2D(x=3, y=3)
3 = {Vector2D@839} Vector2D(x=4, y=4)
4 = {Vector2D@840} Vector2D(x=5, y=5)
5 = {Vector2D@841} Vector2D(x=6, y=6)
```

Improved output of vector class objects in the debug tool window

> It might seem like a good idea to use data classes because they have a toString() method automatically. In the Backward compatibility section of this guide, you'll learn why it's better not to do this.

Consider implementing toString() even if you don't think the class is going to be printed anywhere, as it can help in unexpected ways. For example, inside builders, it may be important to see the current state of the builder.

```
class Person(
    val name: String?,
    val age: Int?,
    val children: List<Person>
) {
    override fun toString(): String =
        "Person(name=$name, age=$age, children=$children)"
}
```

```kotlin
class PersonBuilder {
    var name: String? = null
    var age: Int? = null
    val children = arrayListOf<Person>()
    fun child(personBuilder: PersonBuilder.() -> Unit = {}) {
        children.add(person(personBuilder))
    }
}

fun person(personBuilder: PersonBuilder.() -> Unit = {}): Person {
    val builder = PersonBuilder()
    builder.personBuilder()
    return Person(builder.name, builder.age, builder.children)
}
```

The intended use of the code above is the following:

```
2       person { this: PersonBuilder
3           name = "Pasha"
4           age = 36
5           child { this: PersonBuilder
6               name = "Mark"
7           }
8       }
```

Usage of the person DSL and a breakpoint

If you set a breakpoint on the line after the closing brace of the first child (as on the picture above), you see a non-descriptive string in debug output:

```
> p this = {PersonBuilder@830} PersonBuilder@eed1f14
```

Result of a PersonBuilder debugging

If you add a simple toString() implementation like this:

```kotlin
override fun toString(): String =
    "PersonBuilder(name=$name, age=$age, children=$children)"
```

The debug data becomes much clearer:

```
> p this = {PersonBuilder@830} PersonBuilder(name=Pasha, age=36, children=[Person(name=Mark, age=null, children=[])])
```

You can also see immediately which fields are set and which are not.

> Be careful with exposing fields in toString() because it might be easy to get a StackOverflowException. For example, if children has a reference to a parent, that would create a circular reference. Also, be careful about exposing lists and maps, as toString() can expand a deeply nested hierarchy.

What's next?

Learn about APIs' backward compatibility.

Backward compatibility

This chapter contains considerations about backward compatibility. Here are the "don't do" recommendations:

- Don't add arguments to existing API functions
- Don't use data classes in an API
- Don't make return types narrower

Consider using:

- @PublishedApi annotation
- @RequiresOptIn annotation
- Explicit API mode

Learn about the tools designed to enforce backward compatibility.

Definition of backward compatibility

One of the cornerstones of a good API is backward compatibility. Backward-compatible code allows clients of newer API versions to use the same API code that they used with an older API version. This section describes the main points you should think about to make your API backward-compatible.

There are at least three types of compatibility when talking about APIs:

- Source
- Behavioral
- Binary

Read more about compatibility types

You can count versions of a library as source-compatible when you're sure that your client's application will recompile correctly against a newer version of your library. Usually, it's difficult to implement and check this automatically unless the changes are trivial. In any API, there are always corner cases where source compatibility might be broken by a particular change.

Behavioral compatibility ensures that any new code does not change the semantics of the original code behavior, apart from fixing bugs.

A binary backward-compatible version of a library can replace a previously compiled version of the library. Any software that was compiled against the previous version of the library should continue to work correctly.

It's possible to break binary compatibility without breaking source compatibility, and vice versa.

Some principles of keeping binary backward compatibility are obvious: Don't just remove parts of a public API; instead, deprecate them. The following sections contain lesser-known principles.

"Don't do" recommendations

Don't add arguments to existing API functions

Adding non-default arguments to a public API is a breaking change because the existing code won't have enough information to call the updated methods. Adding even default arguments might also break your users' code.

Breaking backward compatibility is shown below in an example of two classes: lib.kt representing a "library", and client.kt representing a "client" of this "library". This construct for libraries and their clients is common in real-world applications. In this example, the "library" has one function that computes the fifth member of the Fibonacci sequence. The file lib.kt contains:

```
fun fib() = … // Returns the fifth element
```

Let's call this function from another file, client.kt:

```
fun main() {
    println(fib()) // Returns 3
}
```

Let's compile the classes:

```
kotlinc lib.kt client.kt
```

The compilation results in two files: LibKt.class and ClientKt.class.

Let's call the client to make sure that it works:

```
$ kotlin ClientKt.class
3
```

The design is far from perfect and hardcoded for learning purposes. It predefines what element of the sequence you want to obtain, which is incorrect and doesn't follow clean code principles. Let's rewrite it preserving the same default behavior: It will return the fifth element by default, but now it will be possible to provide an element number that you want to get.

lib.kt:

```kotlin
fun fib(numberOfElement: Int = 5) = … // Returns requested member
```

Let's recompile only the "library": kotlinc lib.kt.

Let's run the "client":

```
$ kotlin ClientKt.class
```

The result is:

```
Exception in thread "main" java.lang.NoSuchMethodError: 'int LibKt.fib()'
       at LibKt.main(fib.kt:2)
       at LibKt.main(fib.kt)
       …
```

There is a NoSuchMethodError because the signature of the fib() function changed after compilation.

If you recompile client.kt, it will work again because it will be aware of the new signature. In this example, binary compatibility was broken while preserving source compatibility.

Learn more about what happened with the help of decompilation

> This explanation is JVM-specific.

Let's call javap on the LibKt class before the changes:

```
❯ javap LibKt
Compiled from "lib.kt"
public final class LibKt {
 public static final int fib();
}
```

And after the changes:

```
❯ javap LibKt
Compiled from "lib.kt"
public final class LibKt {
 public static final int fib(int);
 public static int fib$default(int, int, java.lang.Object);
}
```

The method with the signature public static final int fib() was replaced with a new method with the signature public static final int fib(int). At the same time, a proxy method fib$default delegates the execution to fib(int). For the JVM, it's possible to work around this: You need to add a @JvmOverloads annotation. For

multiplatform projects, there is no workaround.

Don't use data classes in an API

Data classes are tempting to use because they are short, concise, and provide some nice functionality out of the box. However, because of some specifics of how data classes work, it's better not to use them in library APIs. Almost any change makes the API not backward-compatible.

Usually, it's difficult to predict how you will need to change a class over time. Even if today you think that it's self-contained, there is no way to be sure that your needs won't change in the future. So, all the issues with data classes only arise when you decide to change such a class.

First, the considerations from the previous section, Don't add arguments to existing API functions, also apply to the constructor as it is also a method. Second, even if you add secondary constructors, it won't solve the compatibility problem. Let's look at the following data class:

```
data class User(
    val name: String,
    val email: String
)
```

For example, over time, you understand that users should go through an activation procedure, so you want to add a new field, "active", with a default value equal to "true". This new field should allow the existing code to work mostly without changes.

As it was already discussed in the section above, you can't just add a new field like this:

```
data class User(
    val name: String,
    val email: String,
    val active: Boolean = true
)
```

Because this change is not binary-compatible.

Let's add a new constructor that accepts only two arguments and calls the primary constructor with a third default argument:

```
data class User(
    val name: String,
    val email: String,
    val active: Boolean = true
) {
    constructor(name: String, email: String) :
            this(name, email, active = true)
}
```

This time there are two constructors, and a signature of one of them matches the constructor of the class before the change:

```
public User(java.lang.String, java.lang.String);
```

But the issue is not with the constructor – it's with the copy function. Its signature has changed from:

```
public final User copy(java.lang.String, java.lang.String);
```

To:

```
public final User copy(java.lang.String, java.lang.String, boolean);
```

And it has made the code binary-incompatible.

Of course, it's possible just to add a property inside the data class, but that would remove all the bonuses of it being a data class. Therefore, it's better not to use data classes in your API because almost any change in them breaks source, binary, or behavioral compatibility.

If you have to use a data class for whatever reason, you have to override the constructor and the copy() method. In addition, if you add a field into the class's body, you have to override the hashCode() and equals() methods.

> It's always an incompatible change to swap the order of arguments because of componentX() methods. It breaks source compatibility and probably will break binary compatibility, too.

Don't make return types narrower

Sometimes, especially when you don't use explicit API mode, a return type declaration can change implicitly. But even if that's not the case, you might want to narrow the signature. For example, you might realize that you need index access to the elements of your collection and want to change the return type from Collection to List. Widening a return type usually breaks source compatibility; for example, converting from List to Collection breaks all the code that uses index access. Narrowing return types is usually a source-compatible change, but it breaks binary compatibility, and this section describes how.

Consider a library function in the library.kt file:

```
public fun x(): Number = 3
```

And an example of its use in the client.kt file:

```
fun main() {
    println(x()) // Prints 3
}
```

Let's compile it with kotlinc library.kt client.kt and make sure that it works:

```
$ kotlin ClientKt
3
```

Let's change the return type of the "library" function x() from Number to Int:

```
fun x(): Int = 3
```

And recompile only the client: kotlinc client.kt. ClientKt doesn't work as expected anymore. It doesn't print 3 and throws an exception instead:

```
Exception in thread "main" java.lang.NoSuchMethodError: 'java.lang.Number Library.x()'
     at ClientKt.main(call.kt:2)
     at ClientKt.main(call.kt)
     …
```

This happens because of the following line in bytecode:

```
0: invokestatic  #12 // Method Library.x:()Ljava/lang/Number;
```

This line means that you call the static method x() returning the type Number. But there is no longer such a method, and so binary compatibility has been violated.

The @PublishedApi annotation

Sometimes, you might need to use a part of your internal API to implement inline functions. You can do this with the @PublishedApi annotation. You should treat parts of code annotated with @PublishedApi as parts of a public API, and, therefore, you should be careful about their backward compatibility.

The @RequiresOptIn annotation

Sometimes, you might want to experiment with your API. In Kotlin, there is a nice way to define that some API is unstable – by using the @RequiresOptIn annotation. However, be aware of the following:

1. If you haven't changed a part of your API for a long time and it's stable, you should reconsider using the @RequiresOptIn annotation.
2. You may use the @RequiresOptIn annotation to define different guarantees to different parts of the API: Preview, Experimental, Internal, Delicate, or Alpha, Beta, RC.
3. You should explicitly define what each level means, write KDoc comments, and add a warning message.

If you depend on an API requiring opt-in, don't use the @OptIn annotation. Instead, use the @RequiresOptIn annotation so that your user is able to consciously choose which API they want to use and which not.

Another example of @RequiresOptIn is when you want to explicitly warn users about the usage of some API. For example, if you maintain a library that utilizes Kotlin reflection, you can annotate classes in this library with @RequiresFullKotlinReflection.

Explicit API mode

You should try to keep your API as transparent as possible. To force the API to be transparent, use the explicit API mode.

Kotlin gives you vast freedom in how you can write code. It is possible to omit type definitions, visibility declarations, or documentation. The explicit API mode forces you as a developer to make implicit things explicit. By the link above, you can find out how to enable it. Let's try to understand why you might need it:

1. Without an explicit API, it's easier to break backward compatibility:

   ```
   // version 1
   fun getToken() = 1

   // version 1.1
   fun getToken() = "1"
   ```

   The return type of getToken() changes, but you don't even need to update the signature for it to break users' code. They expect to get Int, but they get String.

2. The same applies to visibility. If the getToken() function is private, then backward compatibility is not broken. But without an explicit visibility declaration, it's unclear whether an API user should be able to access it. If they should be able to, it should be declared as public and documented; in this case, the change breaks backward compatibility. If not, it should be defined as private or internal, and this change won't be breaking.

Tools designed to enforce backward compatibility

Backward compatibility is a crucial aspect of software development, as it ensures that new versions of a library or framework can be used with existing code without causing any issues. Maintaining backward compatibility can be a difficult and time-consuming task, especially when dealing with a large codebase or complex APIs. It's hard to control it manually, and developers often have to rely on testing and manual inspection to ensure that changes do not break existing code. To address this issue, JetBrains created the Binary compatibility validator, and there is also another solution: japicmp.

> At the moment, both work only for the JVM.

Both solutions have their advantages and disadvantages. japicmp works for any JVM language, and it's both a CLI tool and a build system plugin. However, it requires having both old and new applications packaged as JAR files. It's not that easy to use when you don't have access to older builds of your library. Also, japicmp gives information on changes in Kotlin metadata, which you may not need (because a metadata format is not specified and is considered to be used only for Kotlin internal usage).

The Binary compatibility validator works only as a Gradle plugin, and it is on the Alpha stability level. It doesn't need access to JAR files. It only needs specific dumps of the previous API and the current API. It's capable of collecting these dumps itself. Learn more about these tools below.

Binary compatibility validator

The Binary compatibility validator is a tool that helps you ensure the backward compatibility of your libraries and frameworks by automatically detecting and reporting any breaking changes in the API. The tool analyzes the library's bytecode before and after you made changes and compares the two versions to identify any changes that may break existing code. This makes it easy to detect and fix any issues before they become a problem for your users.

This tool can save a significant amount of time and effort that you would otherwise spend on manual testing and inspection. It can also help prevent issues that may arise due to breaking changes in the API. This can ultimately lead to a better user experience, as users will be able to rely on the stability and compatibility of the library or framework.

japicmp

If you target only the JVM as your platform, you can also use japicmp. japicmp operates on a different level from the Binary compatibility validator: It compares two jar files – old and new – and reports incompatibilities between them.

Be aware that japicmp reports not only incompatibilities but also changes that should not affect a user in any way. For example, consider the following code:

```kotlin
class Calculator {
    fun add(a: Int, b: Int): Int = a + b
    fun multiply(a: Int, b: Int): Int = a * b
}
```

If you add a new method without breaking the compatibility like this:

```kotlin
class Calculator {
    fun add(a: Int, b: Int): Int = a + b
    fun multiply(a: Int, b: Int): Int = a * b
    fun divide(a: Int, b: Int): Int = a / b
}
```

Then japicmp reports the following change:

```
***  MODIFIED CLASS: PUBLIC FINAL Calculator  (not serializable)
	===  CLASS FILE FORMAT VERSION: 52.0 <- 52.0
	===  UNCHANGED SUPERCLASS: java.lang.Object (<- java.lang.Object)
	===  UNCHANGED CONSTRUCTOR: PUBLIC Calculator()
	===  UNCHANGED METHOD: PUBLIC FINAL int add(int, int)
	+++  NEW METHOD: PUBLIC(+) FINAL(+) int divide(int, int)
	===  UNCHANGED METHOD: PUBLIC FINAL int multiply(int, int)
	***  MODIFIED ANNOTATION: kotlin.Metadata
		===  UNCHANGED ELEMENT: xi=48
		===  UNCHANGED ELEMENT: mv=1,8,0
		===  UNCHANGED ELEMENT: k=1
		***  MODIFIED ELEMENT: d1=�#xA;�#xA;�#xA;�#xA;�#xA;� � � � � � � � � �
J� � � � � � � � � �(<-  �#xA;�#xA;�#xA;�#xA;�#xA;� � � � � � � �
� � � � � � � �
		***  MODIFIED ELEMENT: d2=LCalculator;,,()V,add,,a,b,divide,multiply (<- LCalculator;,,()V,add,,a,b,multi
ply)
```

Output of japicmp checking compatibility

It's just a change in the @Metadata annotation, which isn't very interesting, but japicmp is JVM-language agnostic and has to report everything it sees.

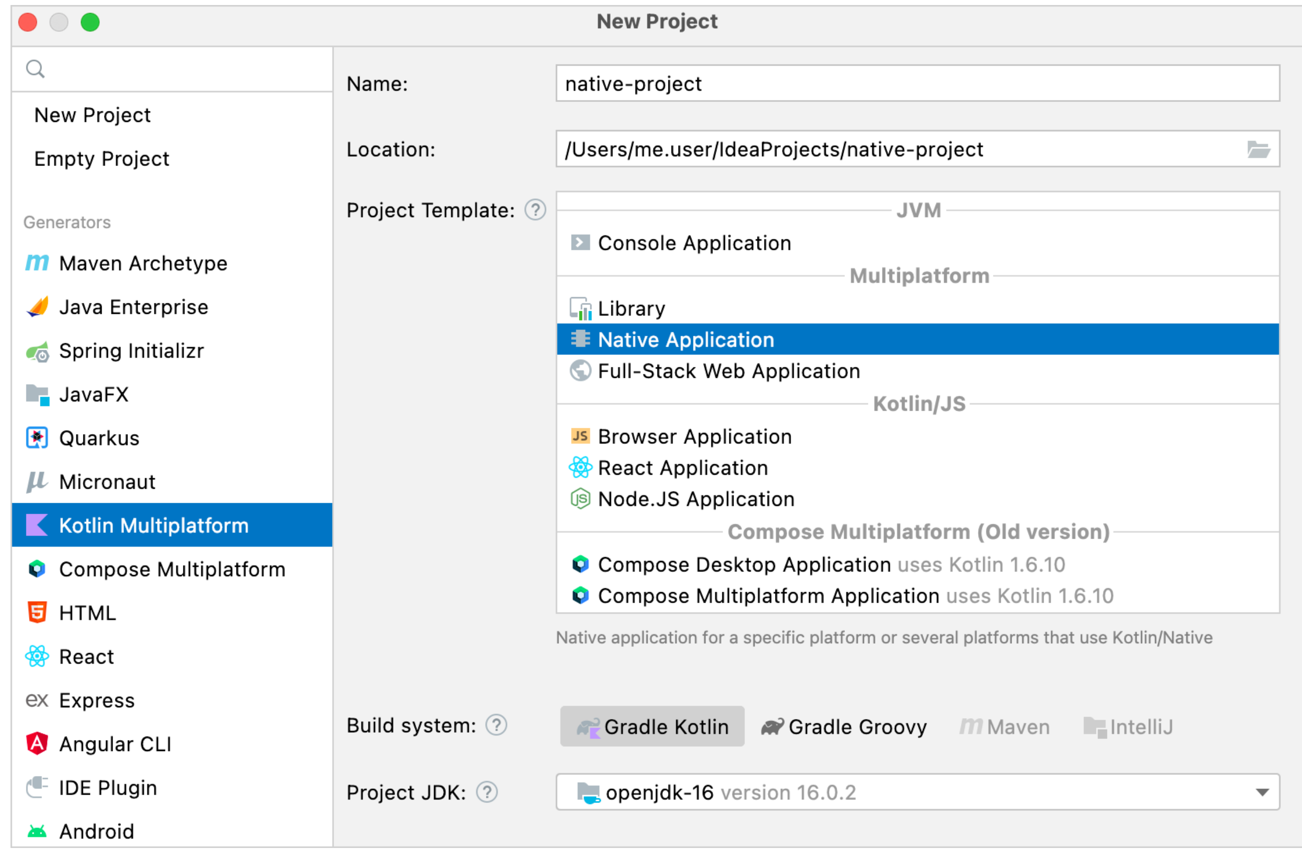
Get started with Kotlin/Native in IntelliJ IDEA

This tutorial demonstrates how to use IntelliJ IDEA for creating a Kotlin/Native application.

To get started, install the latest version of IntelliJ IDEA. The tutorial is applicable to both IntelliJ IDEA Community Edition and the Ultimate Edition.

Create a new Kotlin/Native project in IntelliJ IDEA

1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select Kotlin Multiplatform.
3. Enter a project name, select Native Application as the project template, and click Next.

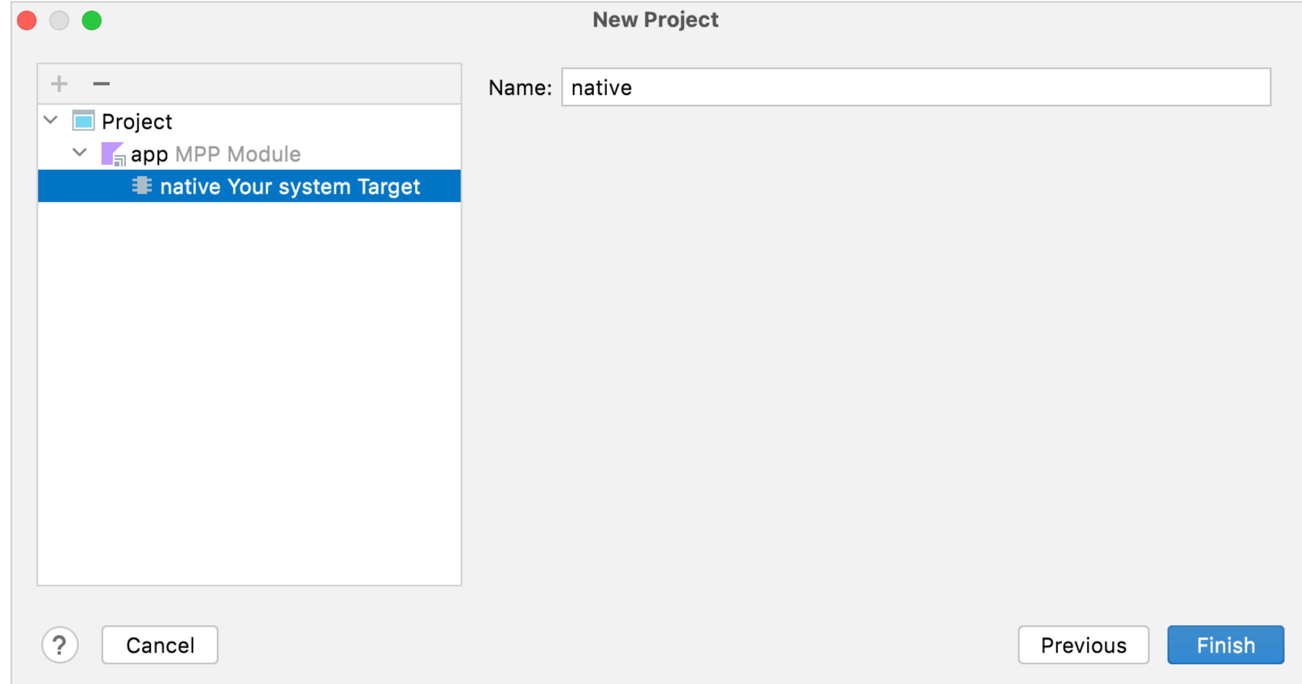


Create a native application

By default, your project will use Gradle with Kotlin DSL as the build system.

> Kotlin/Native doesn't support Maven and IntelliJ IDEA native builder.

4. Accept the default configuration on the next screen and click Finish. Your project will open.



Configure a native application

By default, the wizard creates the necessary Main.kt file with code that prints "Hello, Kotlin/Native!" to the standard output.

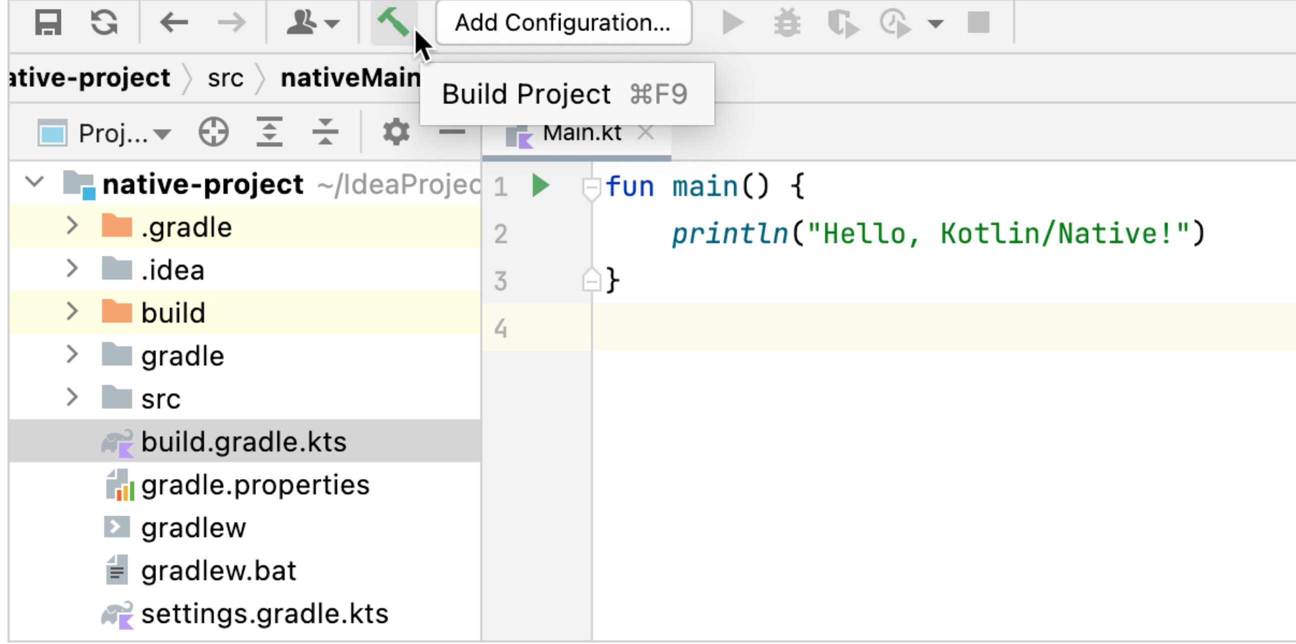
5. Open the build.gradle.kts file, the build script that contains the project settings. To create Kotlin/Native applications, you need the Kotlin Multiplatform Gradle plugin installed. Ensure that you use the latest version of the plugin:

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}
```

- Read more about these settings in the Multiplatform Gradle DSL reference.
- Read more about the Gradle build system in the documentation.

Build and run the application

1. Click Build Project next to the run configuration at the top of the screen:



Build the application

2. In the IntelliJ IDEA terminal or your command-line tool, run the following command:

```
build/bin/native/debugExecutable/<your_app_name>.kexe
```

IntelliJ IDEA prints "Hello, Kotlin/Native!".

You can configure IntelliJ IDEA to build your project automatically:

1. Go to Settings/Preferences | Build, Execution, Deployment | Compiler.
2. On the Compiler page, select Build project automatically.
3. Apply the changes.

Now when you make changes in the class files or save the file (Ctrl + S/Cmd + S), IntelliJ IDEA automatically performs the incremental build of the project.

Update the application

Count the letters in your name

1. Open the file Main.kt in src/nativeMain/kotlin.

   The src directory contains the Kotlin source files and resources. The file Main.kt includes sample code that prints "Hello, Kotlin/Native!" using the println() function.

2. Add code to read the input. Use the readln() function to read the input value and assign it to the name variable:

```kotlin
fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
}
```

3. Eliminate the whitespaces and count the letters:

   - Use the replace() function to remove the empty spaces in the name.
   - Use the scope function let to run the function within the object context.

- Use a string template to insert your name length into the string by adding a dollar sign $ and enclosing it in curly braces – ${it.length}. it is the default name of a lambda parameter.

```kotlin
fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
    // Count the letters in the name.
    name.replace(" ", "").let {
        println("Your name contains ${it.length} letters")
    }
}
```

4. Save the changes and run the following command in the IntelliJ IDEA terminal or your command-line tool:

```
build/bin/native/debugExecutable/<your_app_name>.kexe
```

5. Enter your name and enjoy the result:

```
Terminal:   Local
~/IdeaProjects/native-project  build/bin/native/debugExecutable/native-project.kexe
Hello, enter your name:
John Smith
Your name contains 9 letters
```

Application output

Count the unique letters in your name

1. Open the file Main.kt in src/nativeMain/kotlin.
2. Declare the new extension function countDistinctCharacters() for String:

- Convert the name to lowercase using the lowercase() function.
- Convert the input string to a list of characters using the toList() function.
- Select only the distinct characters in your name using the distinct() function.
- Count the distinct characters using the count() function.

```kotlin
fun String.countDistinctCharacters() = lowercase().toList().distinct().count()
```

3. Use the countDistinctCharacters() function to count the unique letters in your name:

```kotlin
fun String.countDistinctCharacters() = lowercase().toList().distinct().count()

fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
    // Count the letters in the name.
    name.replace(" ", "").let {
        println("Your name contains ${it.length} letters")
        // Print the number of unique letters.
        println("Your name contains ${it.countDistinctCharacters()} unique letters")
    }
}
```

4. Save the changes and run the following command in the IntelliJ IDEA terminal or your command-line tool:

```
build/bin/native/debugExecutable/<your_app_name>.kexe
```

5. Enter your name and enjoy the result:

```
Terminal:   Local   +
~/IdeaProjects/native-project  build/bin/native/debugExecutable/native-project.kexe
Hello, enter your name:
John Smith
Your name contains 9 letters
Your name contains 8 unique letters
```

Application output

What's next?

Once you have created your first application, you can complete our long-form tutorial on Kotlin/Native, Create an app using C Interop and libcurl that explains how to create a native HTTP client and interoperate with C libraries.

Get started with Kotlin/Native using Gradle

Gradle is a build system that is very commonly used in the Java, Android, and other ecosystems. It is the default choice for Kotlin/Native and Multiplatform when it comes to build systems.

While most IDEs, including IntelliJ IDEA, can generate necessary Gradle files, this tutorial covers how to create them manually to provide a better understanding of how things work under the hood.

To get started, install the latest version of Gradle.

> If you would like to use an IDE, check out the Using IntelliJ IDEA tutorial.

Create project files

1. Create a project directory. Inside it, create build.gradle(.kts) Gradle build file with the following content:

Kotlin

```
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "1.9.0"
}

repositories {
    mavenCentral()
}

kotlin {
    macosX64("native") { // on macOS
    // linuxX64("native") // on Linux
    // mingwX64("native") // on Windows
        binaries {
            executable()
        }
    }
}

tasks.withType<Wrapper> {
    gradleVersion = "7.6"
    distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}

repositories {
    mavenCentral()
}

kotlin {
    macosX64('native') { // on macOS
    // linuxX64('native') // on Linux
    // mingwX64('native') // on Windows
        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = '7.6'
    distributionType = 'BIN'
}
```

You can use different target presets, such as macosX64, mingwX64, linuxX64, iosX64, to define the corresponding target platform. The preset name describes a platform for which you are compiling your code. These target presets optionally take the target name as a parameter, which is native in this case. The target name is used to generate the source paths and task names in the project.

2. Create an empty settings.gradle or settings.gradle.kts file in the project directory.

3. Create a directory src/nativeMain/kotlin and place inside the hello.kt file with the following content:

```
fun main() {
    println("Hello, Kotlin/Native!")
}
```

By convention, all sources are located in the src/<target name>[Main|Test]/kotlin directories, where main is for the source code and test is for tests. <target name> corresponds to the target platform (in this case native), as specified in the build file.

Now you are ready to build your project and run the application.

Build and run the application

1. From the root project directory, run the build command:

```
gradle nativeBinaries
```

This command creates the build/bin/native directory with two directories inside: debugExecutable and releaseExecutable. They contain corresponding binary files.

By default, the name of the binary file is the same as the project directory.

2. To run the project, execute the following command:

```
build/bin/native/debugExecutable/<project_name>.kexe
```

Terminal prints "Hello, Kotlin/Native!".

Open the project in an IDE

Now you can open your project in any IDE that supports Gradle. If you use IntelliJ IDEA:

1. Select File | Open....

2. Select the project directory and click Open.

IntelliJ IDEA will automatically detect it as Kotlin/Native project.

> If you face any problem with the project, IntelliJ IDEA will show the error message in the Build tab.

What's next?

Learn how to write Gradle build scripts for real-life Kotlin/Native projects.

Get started with Kotlin/Native using the command-line compiler

Obtain the compiler

The Kotlin/Native compiler is available for macOS, Linux, and Windows. It is available as a command line tool and ships as part of the standard Kotlin distribution and can be downloaded from GitHub Releases. It supports different targets including Linux, macOS, iOS, and others. See the full list of supported targets. While cross-platform compilation is possible, which means using one platform to compile for a different one, in this Kotlin case we'll be targeting the same platform we're compiling on.

While the output of the compiler does not have any dependencies or virtual machine requirements, the compiler itself requires Java 1.8 or higher runtime.

Install the compiler by unpacking its archive to a directory of your choice and adding the path to its /bin directory to the PATH environment variable.

Write "Hello Kotlin/Native" program

The application will print "Hello Kotlin/Native" on the standard output. In a working directory of choice, create a file named hello.kt and enter the following contents:

```
fun main() {
  println("Hello Kotlin/Native!")
}
```

Compile the code from the console

To compile the application use the downloaded compiler to execute the following command:

```
kotlinc-native hello.kt -o hello
```

The value of -o option specifies the name of the output file, so this call should generate a hello.kexe (Linux and macOS) or hello.exe (Windows) binary file. For the full list of available compiler options, see the compiler options reference.

While compilation from the console seems to be easy and clear, it does not scale well for larger projects with hundreds of files and libraries. For real-world projects, it is recommended to use a build system and IDE.

Interoperability with C

Kotlin/Native follows the general tradition of Kotlin to provide excellent existing platform software interoperability. In the case of a native platform, the most important interoperability target is a C library. So Kotlin/Native comes with a cinterop tool, which can be used to quickly generate everything needed to interact with an external library.

The following workflow is expected when interacting with the native library:

1. Create a .def file describing what to include into bindings.
2. Use the cinterop tool to produce Kotlin bindings.
3. Run the Kotlin/Native compiler on an application to produce the final executable.

The interoperability tool analyses C headers and produces a "natural" mapping of the types, functions, and constants into the Kotlin world. The generated stubs can be imported into an IDE for the purpose of code completion and navigation.

Interoperability with Swift/Objective-C is provided too and covered in Objective-C interop.

Platform libraries

Note that in many cases there's no need to use custom interoperability library creation mechanisms described below, as for APIs available on the platform standardized bindings called platform libraries could be used. For example, POSIX on Linux/macOS platforms, Win32 on Windows platform, or Apple frameworks on macOS/iOS are available this way.

Simple example

Install libgit2 and prepare stubs for the git library:

```
cd samples/gitchurn
../../dist/bin/cinterop -def src/nativeInterop/cinterop/libgit2.def \
 -compiler-option -I/usr/local/include -o libgit2
```

Compile the client:

```
../../dist/bin/kotlinc src/gitChurnMain/kotlin \
 -library libgit2 -o GitChurn
```

Run the client:

```
./GitChurn.kexe ../..
```

Create bindings for a new library

To create bindings for a new library, start from creating a .def file. Structurally it's a simple property file, which looks like this:

```
headers = png.h
headerFilter = png.h
package = png
```

Then run the cinterop tool with something like this (note that for host libraries that are not included in the sysroot search paths, headers may be needed):

```
cinterop -def png.def -compiler-option -I/usr/local/include -o png
```

This command will produce a png.klib compiled library and png-build/kotlin directory containing Kotlin source code for the library.

If the behavior for a certain platform needs to be modified, you can use a format like compilerOpts.osx or compilerOpts.linux to provide platform-specific values to the options.

Note that the generated bindings are generally platform-specific, so if you are developing for multiple targets, the bindings need to be regenerated.

After the generation of bindings, they can be used by the IDE as a proxy view of the native library.

For a typical Unix library with a config script, the compilerOpts will likely contain the output of a config script with the --cflags flag (maybe without exact paths).

The output of a config script with --libs will be passed as a -linkedArgs kotlinc flag value (quoted) when compiling.

Select library headers

When library headers are imported to a C program with the #include directive, all of the headers included by these headers are also included in the program. So all header dependencies are included in generated stubs as well.

This behavior is correct but it can be very inconvenient for some libraries. So it is possible to specify in the .def file which of the included headers are to be

imported. The separate declarations from other headers can also be imported in case of direct dependencies.

Filter headers by globs

It is possible to filter headers by globs using filter properties from the .def file. They are treated as a space-separated list of globs.

- To include declarations from headers, use the headerFilter property. If the included header matches any of the globs, the declarations are included in the bindings.

  The globs are applied to the header paths relative to the appropriate include path elements, for example, time.h or curl/curl.h. So if the library is usually included with #include <SomeLibrary/Header.h>, it would probably be correct to filter headers with the following filter:

  ```
  headerFilter = SomeLibrary/**
  ```

  If headerFilter is not provided, all the headers are included. However, we encourage you to use headerFilter and specify the glob as precisely as possible. In this case, the generated library contains only the necessary declarations. It can help avoid various issues when upgrading Kotlin or tools in your development environment.

- To exclude specific headers, use the excludeFilter property.

  It can be helpful to remove redundant or problematic headers and optimize compilation, as declarations from the specified headers are not included into the bindings.

  ```
  excludeFilter = SomeLibrary/time.h
  ```

> If the same header is both included with headerFilter, and excluded with excludeFilter, the latter will have a higher priority. The specified header will not be included into the bindings.

Filter headers by module maps

Some libraries have proper module.modulemap or module.map files in their headers. For example, macOS and iOS system libraries and frameworks do. The module map file describes the correspondence between header files and modules. When the module maps are available, the headers from the modules that are not included directly can be filtered out using the experimental excludeDependentModules option of the .def file:

```
headers = OpenGL/gl.h OpenGL/glu.h GLUT/glut.h
compilerOpts = -framework OpenGL -framework GLUT
excludeDependentModules = true
```

When both excludeDependentModules and headerFilter are used, they are applied as an intersection.

C compiler and linker options

Options passed to the C compiler (used to analyze headers, such as preprocessor definitions) and the linker (used to link final executables) can be passed in the definition file as compilerOpts and linkerOpts respectively. For example:

```
compilerOpts = -DFOO=bar
linkerOpts = -lpng
```

Target-specific options only applicable to the certain target can be specified as well:

```
compilerOpts = -DBAR=bar
compilerOpts.linux_x64 = -DFOO=foo1
compilerOpts.mac_x64 = -DFOO=foo2
```

With such a configuration, C headers will be analyzed with -DBAR=bar -DFOO=foo1 on Linux and with -DBAR=bar -DFOO=foo2 on macOS . Note that any definition file option can have both common and the platform-specific part.

Add custom declarations

Sometimes it is required to add custom C declarations to the library before generating bindings (e.g., for macros). Instead of creating an additional header file with these declarations, you can include them directly to the end of the .def file, after a separating line, containing only the separator sequence ---:

```
headers = errno.h

---

static inline int getErrno() {
    return errno;
}
```

Note that this part of the .def file is treated as part of the header file, so functions with the body should be declared as static. The declarations are parsed after including the files from the headers list.

Include a static library in your klib

Sometimes it is more convenient to ship a static library with your product, rather than assume it is available within the user's environment. To include a static library into .klib use staticLibrary and libraryPaths clauses. For example:

```
headers = foo.h
staticLibraries = libfoo.a
libraryPaths = /opt/local/lib /usr/local/opt/curl/lib
```

When given the above snippet the cinterop tool will search libfoo.a in /opt/local/lib and /usr/local/opt/curl/lib, and if it is found include the library binary into klib.

When using such klib in your program, the library is linked automatically.

Bindings

Basic interop types

All the supported C types have corresponding representations in Kotlin:

- Signed, unsigned integral, and floating point types are mapped to their Kotlin counterpart with the same width.
- Pointers and arrays are mapped to CPointer<T>?.
- Enums can be mapped to either Kotlin enum or integral values, depending on heuristics and the definition file hints.
- Structs and unions are mapped to types having fields available via the dot notation, i.e. someStructInstance.field1.
- typedef are represented as typealias.

Also, any C type has the Kotlin type representing the lvalue of this type, i.e., the value located in memory rather than a simple immutable self-contained value. Think C++ references, as a similar concept. For structs (and typedefs to structs) this representation is the main one and has the same name as the struct itself, for Kotlin enums it is named ${type}Var, for CPointer<T> it is CPointerVar<T>, and for most other types it is ${type}Var.

For types that have both representations, the one with a "lvalue" has a mutable .value property for accessing the value.

Pointer types

The type argument T of CPointer<T> must be one of the "lvalue" types described above, e.g., the C type struct S* is mapped to CPointer<S>, int8_t* is mapped to CPointer<int_8tVar>, and char** is mapped to CPointer<CPointerVar<ByteVar>>.

C null pointer is represented as Kotlin's null, and the pointer type CPointer<T> is not nullable, but the CPointer<T>? is. The values of this type support all the Kotlin operations related to handling null, e.g. ?:, ?., !! etc.:

```
val path = getenv("PATH")?.toKString() ?: ""
```

Since the arrays are also mapped to CPointer<T>, it supports the [] operator for accessing values by index:

```
fun shift(ptr: CPointer<BytePtr>, length: Int) {
    for (index in 0 .. length - 2) {
        ptr[index] = ptr[index + 1]
    }
}
```

The .pointed property for CPointer<T> returns the lvalue of type T, pointed by this pointer. The reverse operation is .ptr: it takes the lvalue and returns the pointer to

it.

void* is mapped to COpaquePointer – the special pointer type which is the supertype for any other pointer type. So if the C function takes void*, then the Kotlin binding accepts any CPointer.

Casting a pointer (including COpaquePointer) can be done with .reinterpret<T>, e.g.:

```
val intPtr = bytePtr.reinterpret<IntVar>()
```

or

```
val intPtr: CPointer<IntVar> = bytePtr.reinterpret()
```

As is with C, these reinterpret casts are unsafe and can potentially lead to subtle memory problems in the application.

Also, there are unsafe casts between CPointer<T>? and Long available, provided by the .toLong() and .toCPointer<T>() extension methods:

```
val longValue = ptr.toLong()
val originalPtr = longValue.toCPointer<T>()
```

Note that if the type of the result is known from the context, the type argument can be omitted as usual due to the type inference.

Memory allocation

The native memory can be allocated using the NativePlacement interface, e.g.

```
val byteVar = placement.alloc<ByteVar>()
```

or

```
val bytePtr = placement.allocArray<ByteVar>(5)
```

The most "natural" placement is in the object nativeHeap. It corresponds to allocating native memory with malloc and provides an additional .free() operation to free allocated memory:

```
val buffer = nativeHeap.allocArray<ByteVar>(size)
<use buffer>
nativeHeap.free(buffer)
```

However, the lifetime of allocated memory is often bound to the lexical scope. It is possible to define such scope with memScoped { ... }. Inside the braces, the temporary placement is available as an implicit receiver, so it is possible to allocate native memory with alloc and allocArray, and the allocated memory will be automatically freed after leaving the scope.

For example, the C function returning values through pointer parameters can be used like

```
val fileSize = memScoped {
    val statBuf = alloc<stat>()
    val error = stat("/", statBuf.ptr)
    statBuf.st_size
}
```

Pass pointers to bindings

Although C pointers are mapped to the CPointer<T> type, the C function pointer-typed parameters are mapped to CValuesRef<T>. When passing CPointer<T> as the value of such a parameter, it is passed to the C function as is. However, the sequence of values can be passed instead of a pointer. In this case the sequence is passed "by value", i.e., the C function receives the pointer to the temporary copy of that sequence, which is valid only until the function returns.

The CValuesRef<T> representation of pointer parameters is designed to support C array literals without explicit native memory allocation. To construct the immutable self-contained sequence of C values, the following methods are provided:

- ${type}Array.toCValues(), where type is the Kotlin primitive type
- Array<CPointer<T>?>.toCValues(), List<CPointer<T>?>.toCValues()

- cValuesOf(vararg elements: ${type}), where type is a primitive or pointer

For example:

C:

```
void foo(int* elements, int count);
...
int elements[] = {1, 2, 3};
foo(elements, 3);
```

Kotlin:

```
foo(cValuesOf(1, 2, 3), 3)
```

Strings

Unlike other pointers, the parameters of type const char* are represented as a Kotlin String. So it is possible to pass any Kotlin string to a binding expecting a C string.

There are also some tools available to convert between Kotlin and C strings manually:

- fun CPointer<ByteVar>.toKString(): String
- val String.cstr: CValuesRef<ByteVar>.

To get the pointer, .cstr should be allocated in native memory, e.g.

val cString = kotlinString.cstr.getPointer(nativeHeap)

In all cases, the C string is supposed to be encoded as UTF-8.

To skip automatic conversion and ensure raw pointers are used in the bindings, a noStringConversion statement in the .def file could be used, i.e.

```
noStringConversion = LoadCursorA LoadCursorW
```

This way any value of type CPointer<ByteVar> can be passed as an argument of const char* type. If a Kotlin string should be passed, code like this could be used:

```
memScoped {
    LoadCursorA(null, "cursor.bmp".cstr.ptr)   // for ASCII version
    LoadCursorW(null, "cursor.bmp".wcstr.ptr)  // for Unicode version
}
```

Scope-local pointers

It is possible to create a scope-stable pointer of C representation of CValues<T> instance using the CValues<T>.ptr extension property, available under memScoped { ... }. It allows using the APIs which require C pointers with a lifetime bound to a certain MemScope. For example:

```
memScoped {
    items = arrayOfNulls<CPointer<ITEM>?>(6)
    arrayOf("one", "two").forEachIndexed { index, value -> items[index] = value.cstr.ptr }
    menu = new_menu("Menu".cstr.ptr, items.toCValues().ptr)
    ...
}
```

In this example, all values passed to the C API new_menu() have a lifetime of the innermost memScope it belongs to. Once the control flow leaves the memScoped scope the C pointers become invalid.

Pass and receive structs by value

When a C function takes or returns a struct / union T by value, the corresponding argument type or return type is represented as CValue<T>.

CValue<T> is an opaque type, so the structure fields cannot be accessed with the appropriate Kotlin properties. It should be possible, if an API uses structures as handles, but if field access is required, there are the following conversion methods available:

- fun T.readValue(): CValue<T>. Converts (the lvalue) T to a CValue<T>. So to construct the CValue<T>, T can be allocated, filled, and then converted to CValue<T>.

- CValue<T>.useContents(block: T.() -> R): R. Temporarily places the CValue<T> to memory, and then runs the passed lambda with this placed value T as receiver. So to read a single field, the following code can be used:

  ```
  val fieldValue = structValue.useContents { field }
  ```

Callbacks

To convert a Kotlin function to a pointer to a C function, staticCFunction(::kotlinFunction) can be used. It is also able to provide the lambda instead of a function reference. The function or lambda must not capture any values.

Pass user data to callbacks

Often C APIs allow passing some user data to callbacks. Such data is usually provided by the user when configuring the callback. It is passed to some C function (or written to the struct) as e.g. void*. However, references to Kotlin objects can't be directly passed to C. So they require wrapping before configuring the callback and then unwrapping in the callback itself, to safely swim from Kotlin to Kotlin through the C world. Such wrapping is possible with StableRef class.

To wrap the reference:

```
val stableRef = StableRef.create(kotlinReference)
val voidPtr = stableRef.asCPointer()
```

where the voidPtr is a COpaquePointer and can be passed to the C function.

To unwrap the reference:

```
val stableRef = voidPtr.asStableRef<KotlinClass>()
val kotlinReference = stableRef.get()
```

where kotlinReference is the original wrapped reference.

The created StableRef should eventually be manually disposed using the .dispose() method to prevent memory leaks:

```
stableRef.dispose()
```

After that it becomes invalid, so voidPtr can't be unwrapped anymore.

See the samples/libcurl for more details.

Macros

Every C macro that expands to a constant is represented as a Kotlin property. Other macros are not supported. However, they can be exposed manually by wrapping them with supported declarations. E.g. function-like macro FOO can be exposed as function foo by adding the custom declaration to the library:

```
headers = library/base.h

---

static inline int foo(int arg) {
    return FOO(arg);
}
```

Definition file hints

The .def file supports several options for adjusting the generated bindings.

- excludedFunctions property value specifies a space-separated list of the names of functions that should be ignored. This may be required because a function declared in the C header is not generally guaranteed to be really callable, and it is often hard or impossible to figure this out automatically. This option can also be used to workaround a bug in the interop itself.

- strictEnums and nonStrictEnums properties values are space-separated lists of the enums that should be generated as a Kotlin enum or as integral values correspondingly. If the enum is not included into any of these lists, then it is generated according to the heuristics.

- noStringConversion property value is space-separated lists of the functions whose const char* parameters shall not be auto-converted as Kotlin string

Portability

Sometimes the C libraries have function parameters or struct fields of a platform-dependent type, e.g. long or size_t. Kotlin itself doesn't provide neither implicit integer casts nor C-style integer casts (e.g. (size_t) intValue), so to make writing portable code in such cases easier, the convert method is provided:

```
fun ${type1}.convert<${type2}>(): ${type2}
```

where each of type1 and type2 must be an integral type, either signed or unsigned.

.convert<${type}> has the same semantics as one of the .toByte, .toShort, .toInt, .toLong, .toUByte, .toUShort, .toUInt or .toULong methods, depending on type.

The example of using convert:

```
fun zeroMemory(buffer: COpaquePointer, size: Int) {
    memset(buffer, 0, size.convert<size_t>())
}
```

Also, the type parameter can be inferred automatically and so may be omitted in some cases.

Object pinning

Kotlin objects could be pinned, i.e. their position in memory is guaranteed to be stable until unpinned, and pointers to such objects inner data could be passed to the C functions. For example

```
fun readData(fd: Int): String {
    val buffer = ByteArray(1024)
    buffer.usePinned { pinned ->
        while (true) {
            val length = recv(fd, pinned.addressOf(0), buffer.size.convert(), 0).toInt()

            if (length <= 0) {
               break
            }
            // Now `buffer` has raw data obtained from the `recv()` call.
        }
    }
}
```

Here we use service function usePinned, which pins an object, executes block and unpins it on normal and exception paths.

Mapping primitive data types from C – tutorial

In this tutorial, you will learn what C data types are visible in Kotlin/Native and vice versa. You will:

- See what Data types are in C language.
- Create a tiny C Library that uses those types in exports.
- Inspect generated Kotlin APIs from a C library.
- Find how Primitive types in Kotlin are mapped to C.

Types in C language

What types are there in the C language? Let's take the C data types article from Wikipedia as a basis. There are following types in the C programming language:

- basic types char, int, float, double with modifiers signed, unsigned, short, long
- structures, unions, arrays
- pointers

- function pointers

There are also more specific types:

- boolean type (from C99)
- size_t and ptrdiff_t (also ssize_t)
- fixed width integer types, such as int32_t or uint64_t (from C99)

There are also the following type qualifiers in the C language: const, volatile, restruct, atomic.

The best way to see what C data types are visible in Kotlin is to try it.

Example C library

Create a lib.h file to see how C functions are mapped into Kotlin:

```
#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void ints(char c, short d, int e, long f);
void uints(unsigned char c, unsigned short d, unsigned int e, unsigned long f);
void doubles(float a, double b);

#endif
```

The file is missing the extern "C" block, which is not needed for this example, but may be necessary if you use C++ and overloaded functions. The C++ compatibility thread on Stackoverflow contains more details on this.

For every set of .h files, you will be using the cinterop tool from Kotlin/Native to generate a Kotlin/Native library, or .klib. The generated library will bridge calls from Kotlin/Native to C. It includes respective Kotlin declarations for the definitions form the .h files. It is only necessary to have a .h file to run the cinterop tool. And you do not need to create a lib.c file, unless you want to compile and run the example. More details on this are covered in the C interop page. It is enough for the tutorial to create the lib.def file with the following content:

```
headers = lib.h
```

You may include all declarations directly into the .def file after a --- separator. It can be helpful to include macros or other C defines into the code generated by the cinterop tool. Method bodies are compiled and fully included into the binary too. Use that feature to have a runnable example without a need for a C compiler. To implement that, you need to add implementations to the C functions from the lib.h file, and place these functions into a .def file. You will have the following interop.def result:

```

---

void ints(char c, short d, int e, long f) { }
void uints(unsigned char c, unsigned short d, unsigned int e, unsigned long f) { }
void doubles(float a, double b) { }
```

The interop.def file is enough to compile and run the application or open it in an IDE. Now it is time to create project files, open the project in IntelliJ IDEA and run it.

Inspect generated Kotlin APIs for a C library

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the Gradle build system through the kotlin-multiplatform plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the A Basic Kotlin/Native Application tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and multiplatform builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64("native") { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64("native") { // on Windows
    val main by compilations.getting
    val interop by main.cinterops.creating

    binaries {
      executable()
    }
  }
}

tasks.wrapper {
  gradleVersion = "7.6"
  distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64('native') { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64('native') { // on Windows
    compilations.main.cinterops {
      interop
    }

    binaries {
      executable()
    }
  }
}

wrapper {
  gradleVersion = '7.6'
  distributionType = 'BIN'
}
```

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the kotlin-multiplatform plugin documentation to learn about all the different ways you could configure it.

Create a src/nativeMain/kotlin/hello.kt stub file with the following content to see how C primitive type declarations are visible from Kotlin:

```
import interop.*

fun main() {
  println("Hello Kotlin/Native!")

  ints(/* fix me*/)
```

```
  uints(/* fix me*/)
  doubles(/* fix me*/)
}
```

Now you are ready to open the project in IntelliJ IDEA and to see how to fix the example project. While doing that, see how C primitive types are mapped into Kotlin/Native.

Primitive types in kotlin

With the help of IntelliJ IDEA's Go to | Declaration or compiler errors, you see the following generated API for the C functions:

```
fun ints(c: Byte, d: Short, e: Int, f: Long)
fun uints(c: UByte, d: UShort, e: UInt, f: ULong)
fun doubles(a: Float, b: Double)
```

C types are mapped in the way we would expect, note that char type is mapped to kotlin.Byte as it is usually an 8-bit signed value.

C	Kotlin
char	kotlin.Byte
unsigned char	kotlin.UByte
short	kotlin.Short
unsigned short	kotlin.UShort
int	kotlin.Int
unsigned int	kotlin.UInt
long long	kotlin.Long
unsigned long long	kotlin.ULong
float	kotlin.Float
double	kotlin.Double

Fix the code

You've seen all definitions and it is the time to fix the code. Run the runDebugExecutableNative Gradle task in IDE or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

The final code in the hello.kt file may look like that:

```
import interop.*
```

```
fun main() {
  println("Hello Kotlin/Native!")

  ints(1, 2, 3, 4)
  uints(5, 6, 7, 8)
  doubles(9.0f, 10.0)
}
```

Next steps

Continue to explore more complicated C language types and their representation in Kotlin/Native in the next tutorials:

- Mapping struct and union types from C
- Mapping function pointers from C
- Mapping strings from C

The C interop documentation covers more advanced scenarios of the interop.

Mapping struct and union types from C – tutorial

This is the second post in the series. The very first tutorial of the series is Mapping primitive data types from C. There are also the Mapping function pointers from C and Mapping Strings from C tutorials.

In the tutorial, you will learn:

- How struct and union types are mapped
- How to use struct and union type from Kotlin

Mapping struct and union C types

The best way to understand the mapping between Kotlin and C is to try a tiny example. We will declare a struct and a union in the C language, to see how they are mapped into Kotlin.

Kotlin/Native comes with the cinterop tool, the tool generates bindings between the C language and Kotlin. It uses a .def file to specify a C library to import. More details are discussed in the Interop with C Libraries tutorial.

In the previous tutorial, you've created a lib.h file. This time, include those declarations directly into the interop.def file, after the --- separator line:

```
---

typedef struct {
  int a;
  double b;
} MyStruct;

void struct_by_value(MyStruct s) {}
void struct_by_pointer(MyStruct* s) {}

typedef union {
  int a;
  MyStruct b;
  float c;
} MyUnion;

void union_by_value(MyUnion u) {}
void union_by_pointer(MyUnion* u) {}
```

The interop.def file is enough to compile and run the application or open it in an IDE. Now it is time to create project files, open the project in IntelliJ IDEA and run it.

Inspect Generated Kotlin APIs for a C library

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the Gradle build system through the kotlin-multiplatform plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the A Basic Kotlin/Native Application tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and multiplatform builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64("native") { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64("native") { // on Windows
    val main by compilations.getting
    val interop by main.cinterops.creating

    binaries {
      executable()
    }
  }
}

tasks.wrapper {
  gradleVersion = "7.6"
  distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64('native') { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64('native') { // on Windows
    compilations.main.cinterops {
      interop
    }

    binaries {
      executable()
    }
  }
}

wrapper {
  gradleVersion = '7.6'
  distributionType = 'BIN'
}
```

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the kotlin-multiplatform plugin documentation to learn about all the different ways you could configure it.

Create a src/nativeMain/kotlin/hello.kt stub file with the following content to see how C struct and union declarations are visible from Kotlin:

```
import interop.*

fun main() {
  println("Hello Kotlin/Native!")

  struct_by_value(/* fix me*/)
  struct_by_pointer(/* fix me*/)
  union_by_value(/* fix me*/)
  union_by_pointer(/* fix me*/)
}
```

Now you are ready to open the project in IntelliJ IDEA and to see how to fix the example project. While doing that, see how C struct and union types are mapped into Kotlin/Native.

Struct and union types in Kotlin

With the help of IntelliJ IDEA's Go to | Declaration or compiler errors, you see the following generated API for the C functions, struct, and union:

```
fun struct_by_value(s: CValue<MyStruct>)
fun struct_by_pointer(s: CValuesRef<MyStruct>?)

fun union_by_value(u: CValue<MyUnion>)
fun union_by_pointer(u: CValuesRef<MyUnion>?)

class MyStruct constructor(rawPtr: NativePtr /* = NativePtr */) : CStructVar {
    var a: Int
    var b: Double
    companion object : CStructVar.Type
}

class MyUnion constructor(rawPtr: NativePtr /* = NativePtr */) : CStructVar {
    var a: Int
    val b: MyStruct
    var c: Float
    companion object : CStructVar.Type
}
```

You see that cinterop generated wrapper types for our struct and union types. For MyStruct and MyUnion type declarations in C, there are the Kotlin classes MyStruct and MyUnion generated respectively. The wrappers inherit from the CStructVar base class and declare all fields as Kotlin properties. It uses CValue<T> to represent a by-value structure parameter and CValuesRef<T>? to represent passing a pointer to a structure or a union.

Technically, there is no difference between struct and union types on the Kotlin side. Note that a, b, and c properties of MyUnion class in Kotlin use the same memory location to read/write their value just like union does in C language.

More details and advanced use-cases are presented in the
C Interop documentation

Use struct and union types from Kotlin

It is easy to use the generated wrapper classes for C struct and union types from Kotlin. Thanks to the generated properties, it feels natural to use them in Kotlin code. The only question, so far, is how to create a new instance on those classes. As you see from the declarations of MyStruct and MyUnion, their constructors require a NativePtr. Of course, you are not willing to deal with pointers manually. Instead, you can use Kotlin API to have those objects instantiated for us.

Let's take a look at the generated functions that take our MyStruct and MyUnion as parameters. You see that by-value parameters are represented as kotlinx.cinterop.CValue<T>. And for typed pointer parameters you see kotlinx.cinterop.CValuesRef<T>. Kotlin provides us with an API to deal with both types easily, let's try it and see.

Create a CValue

CValue<T> type is used to pass by-value parameters to a C function call. Use cValue function to create CValue<T> object instance. The function requires a lambda

function with a receiver to initialize the underlying C type in-place. The function is declared as follows:

```
fun <reified T : CStructVar> cValue(initialize: T.() -> Unit): CValue<T>
```

Now it is time to see how to use cValue and pass by-value parameters:

```
fun callValue() {

  val cStruct = cValue<MyStruct> {
    a = 42
    b = 3.14
  }
  struct_by_value(cStruct)

  val cUnion = cValue<MyUnion> {
    b.a = 5
    b.b = 2.7182
  }

  union_by_value(cUnion)
}
```

Create struct and union as CValuesRef

CValuesRef<T> type is used in Kotlin to pass a typed pointer parameter of a C function. First, you need an instance of MyStruct and MyUnion classes. Create them directly in the native memory. Use the

```
fun <reified T : kotlinx.cinterop.CVariable> alloc(): T
```

extension function on kotlinx.cinterop.NativePlacement type for this.

NativePlacement represents native memory with functions similar to malloc and free. There are several implementations of NativePlacement. The global one is called with kotlinx.cinterop.nativeHeap and don't forget to call the nativeHeap.free(..) function to free the memory after use.

Another option is to use the

```
fun <R> memScoped(block: kotlinx.cinterop.MemScope.() -> R): R
```

function. It creates a short-lived memory allocation scope, and all allocations will be cleaned up automatically at the end of the block.

Your code to call functions with pointers will look like this:

```
fun callRef() {
  memScoped {
    val cStruct = alloc<MyStruct>()
    cStruct.a = 42
    cStruct.b = 3.14

    struct_by_pointer(cStruct.ptr)

val cUnion = alloc<MyUnion>()
    cUnion.b.a = 5
    cUnion.b.b = 2.7182

    union_by_pointer(cUnion.ptr)
  }
}
```

Note that this code uses the extension property ptr which comes from a memScoped lambda receiver type, to turn MyStruct and MyUnion instances into native pointers.

The MyStruct and MyUnion classes have the pointer to the native memory underneath. The memory will be released when a memScoped function ends, which is equal to the end of its block. Make sure that a pointer is not used outside of the memScoped call. You may use Arena() or nativeHeap for pointers that should be available longer, or are cached inside a C library.

Conversion between CValue and CValuesRef

Of course, there are use cases when you need to pass a struct as a value to one call, and then, to pass the same struct as a reference to another call. This is possible in Kotlin/Native too. A NativePlacement will be needed here.

Let's see now CValue<T> is turned to a pointer first:

```
fun callMix_ref() {
  val cStruct = cValue<MyStruct> {
    a = 42
    b = 3.14
  }

  memScoped {
    struct_by_pointer(cStruct.ptr)
  }
}
```

This code uses the extension property ptr which comes from memScoped lambda receiver type to turn MyStruct and MyUnion instances into native pointers. Those pointers are only valid inside the memScoped block.

For the opposite conversion, to turn a pointer into a by-value variable, we call the readValue() extension function:

```
fun callMix_value() {
  memScoped {
    val cStruct = alloc<MyStruct>()
    cStruct.a = 42
    cStruct.b = 3.14

    struct_by_value(cStruct.readValue())
  }
}
```

Run the code

Now when you have learned how to use C declarations in your code, you are ready to try it out on a real example. Let's fix the code and see how it runs by calling the runDebugExecutableNative Gradle task in the IDE or by using the following console command:

```
./gradlew runDebugExecutableNative
```

The final code in the hello.kt file may look like this:

```
import interop.*
import kotlinx.cinterop.alloc
import kotlinx.cinterop.cValue
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.ptr
import kotlinx.cinterop.readValue

fun main() {
  println("Hello Kotlin/Native!")

  val cUnion = cValue<MyUnion> {
    b.a = 5
    b.b = 2.7182
  }

  memScoped {
    union_by_value(cUnion)
    union_by_pointer(cUnion.ptr)
  }

  memScoped {
    val cStruct = alloc<MyStruct> {
      a = 42
      b = 3.14
    }

    struct_by_value(cStruct.readValue())
    struct_by_pointer(cStruct.ptr)
  }
}
```

Next steps

Continue exploring the C language types and their representation in Kotlin/Native in the related tutorials:

- Mapping primitive data types from C
- Mapping function pointers from C
- Mapping strings from C

The C Interop documentation covers more advanced scenarios of the interop.

Mapping function pointers from C – tutorial

This is the third post in the series. The very first tutorial is Mapping primitive data types from C. There are also Mapping struct and union types from C and Mapping strings from C tutorials.

In this tutorial We will learn how to:

- Pass Kotlin function as C function pointer
- Use C function pointer from Kotlin

Mapping function pointer types from C

The best way to understand the mapping between Kotlin and C is to try a tiny example. Declare a function that accepts a function pointer as a parameter and another function that returns a function pointer.

Kotlin/Native comes with the cinterop tool; the tool generates bindings between the C language and Kotlin. It uses a .def file to specify a C library to import. More details on this are in Interop with C Libraries.

The quickest way to try out C API mapping is to have all C declarations in the interop.def file, without creating any .h of .c files at all. Then place the C declarations in a .def file after the special --- separator line:

```
---

int myFun(int i) {
  return i+1;
}

typedef int  (*MyFun)(int);

void accept_fun(MyFun f) {
  f(42);
}

MyFun supply_fun() {
  return myFun;
}
```

The interop.def file is enough to compile and run the application or open it in an IDE. Now it is time to create project files, open the project in IntelliJ IDEA and run it.

Inspect generated Kotlin APIs for a C library

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the Gradle build system through the kotlin-multiplatform plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the A Basic Kotlin/Native Application tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and multiplatform builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64("native") { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64("native") { // on Windows
    val main by compilations.getting
    val interop by main.cinterops.creating

    binaries {
      executable()
    }
  }
}

tasks.wrapper {
  gradleVersion = "7.6"
  distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64('native') { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64('native') { // on Windows
    compilations.main.cinterops {
      interop
    }

    binaries {
      executable()
    }
  }
}

wrapper {
  gradleVersion = '7.6'
  distributionType = 'BIN'
}
```

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the kotlin-multiplatform plugin documentation to learn about all the different ways you could configure it.

Let's create a src/nativeMain/kotlin/hello.kt stub file with the following content to see how C function pointer declarations are visible from Kotlin:

```
import interop.*
```

```
fun main() {
  println("Hello Kotlin/Native!")

  accept_fun(https://kotlinlang.org/*fix me */)
  val useMe = supply_fun()
}
```

Now you are ready to open the project in IntelliJ IDEA and to see how to fix the example project. While doing that, see how C functions are mapped into Kotlin/Native declarations.

C function pointers in Kotlin

With the help of IntelliJ IDEA's Go To | Declaration or Usages or compiler errors, see the following declarations for the C functions:

```
fun accept_fun(f: MyFun? /* = CPointer<CFunction<(Int) -> Int>>? */)
fun supply_fun(): MyFun? /* = CPointer<CFunction<(Int) -> Int>>? */

fun myFun(i: kotlin.Int): kotlin.Int

typealias MyFun = kotlinx.cinterop.CPointer<kotlinx.cinterop.CFunction<(kotlin.Int) -> kotlin.Int>>

typealias MyFunVar = kotlinx.cinterop.CPointerVarOf<lib.MyFun>
```

You see that the function's typedef from C has been turned into Kotlin typealias. It uses CPointer<..> type to represent the pointer parameters, and CFunction<(Int)->Int> to represent the function signature. There is an invoke operator extension function available for all CPointer<CFunction<..> types, so that it is possible to call it as you would call any other function in Kotlin.

Pass Kotlin function as C function pointer

It is the time to try using C functions from the Kotlin program. Call the accept_fun function and pass the C function pointer to a Kotlin lambda:

```
fun myFun() {
  accept_fun(staticCFunction<Int, Int> { it + 1 })
}
```

This call uses the staticCFunction{..} helper function from Kotlin/Native to wrap a Kotlin lambda function into a C function pointer. It only allows having unbound and non-capturing lambda functions. For example, it is not able to use a local variable from the function. You may only use globally visible declarations. Throwing exceptions from a staticCFunction{..} will end up in non-deterministic side-effects. It is vital to make sure that you code is not throwing any sudden exceptions from it.

Use the C function pointer from Kotlin

The next step is to call a C function pointer from a C pointer that you have from the supply_fun() call:

```
fun myFun2() {
  val functionFromC = supply_fun() ?: error("No function is returned")

  functionFromC(42)
}
```

Kotlin turns the function pointer return type into a nullable CPointer<CFunction<..> object. There is the need to explicitly check for null first. The elvis operator for that in the code above. The cinterop tool helps us to turn a C function pointer into an easy to call object in Kotlin. This is what we did on the last line.

Fix the code

You've seen all definitions and it is time to fix and run the code. Run the runDebugExecutableNative Gradle task in the IDE or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

The code in the hello.kt file may look like this:

```
import interop.*
import kotlinx.cinterop.*

fun main() {
  println("Hello Kotlin/Native!")

  val cFunctionPointer = staticCFunction<Int, Int> { it + 1 }
  accept_fun(cFunctionPointer)

  val funFromC = supply_fun() ?: error("No function is returned")
  funFromC(42)
}
```

Next Steps

Continue exploring more C language types and their representation in Kotlin/Native in next tutorials:

- Mapping primitive data types from C
- Mapping struct and union types from C
- Mapping strings from C

The C Interop documentation covers more advanced scenarios of the interop.

Mapping Strings from C – tutorial

This is the last tutorial in the series. The first tutorial of the series is Mapping primitive data types from C. There are also Mapping struct and union types from C and Mapping function pointers from C tutorials.

In this tutorial, you'll see how to deal with C strings in Kotlin/Native. You will learn how to:

- Pass a Kotlin string to C
- Read a C string in Kotlin
- Receive C string bytes into a Kotlin string

Working with C strings

There is no dedicated type in C language for strings. A developer knows from a method signature or the documentation, whether a given char * means a C string in the context. Strings in the C language are null-terminated, a trailing zero character \0 is added at the end of a bytes sequence to mark a string termination. Usually, UTF-8 encoded strings are used. The UTF-8 encoding uses variable width characters, and it is backward compatible with ASCII. Kotlin/Native uses UTF-8 character encoding by default.

The best way to understand the mapping between C and Kotlin languages is to try it out on a small example. Create a small library headers for that. First, create a lib.h file with the following declaration of functions that deal with the C strings:

```
#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void pass_string(char* str);
char* return_string();
int copy_string(char* str, int size);

#endif
```

In the example, you see the most popular ways to pass or receive a string in the C language. Take the return of return_string with care. In general, it is best to make sure you use the right function to dispose the returned char* with the right free(..) function call.

Kotlin/Native comes with the cinterop tool; the tool generates bindings between the C language and Kotlin. It uses a .def file to specify a C library to import. More details on this are in the Interop with C Libraries tutorial. The quickest way to try out C API mapping is to have all C declarations in the interop.def file, without

creating any .h of .c files at all. Then place the C declarations in a interop.def file after the special --- separator line:

```
headers = lib.h
---

void pass_string(char* str) {
}

char* return_string() {
  return "C string";
}

int copy_string(char* str, int size) {
  *str++ = 'C';
  *str++ = ' ';
  *str++ = 'K';
  *str++ = '/';
  *str++ = 'N';
  *str++ = 0;
  return 0;
}
```

The interop.def file is enough to compile and run the application or open it in an IDE. Now it is time to create project files, open the project in IntelliJ IDEA and run it.

Inspect generated Kotlin APIs for a C library

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the Gradle build system through the kotlin-multiplatform plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the A Basic Kotlin/Native Application tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and multiplatform builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64("native") { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64("native") { // on Windows
    val main by compilations.getting
    val interop by main.cinterops.creating

    binaries {
      executable()
    }
  }
}

tasks.wrapper {
  gradleVersion = "7.6"
  distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64('native') { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64('native') { // on Windows
    compilations.main.cinterops {
      interop
    }

    binaries {
      executable()
    }
  }
}

wrapper {
  gradleVersion = '7.6'
  distributionType = 'BIN'
}
```

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the kotlin-multiplatform plugin documentation to learn about all the different ways you could configure it.

Let's create a src/nativeMain/kotlin/hello.kt stub file with the following content to see how C string declarations are visible from Kotlin:

```
import interop.*

fun main() {
  println("Hello Kotlin/Native!")

  pass_string(/*fix me*/)
  val useMe = return_string()
  val useMe2 = copy_string(/*fix me*/)
}
```

Now you are ready to open the project in IntelliJ IDEA and to see how to fix the example project. While doing that, see how C strings are mapped into Kotlin/Native.

Strings in Kotlin

With the help of IntelliJ IDEA's Go to | Declaration or compiler errors, you see the following generated API for the C functions:

```
fun pass_string(str: CValuesRef<ByteVar /* = ByteVarOf<Byte> */>?)
fun return_string(): CPointer<ByteVar /* = ByteVarOf<Byte> */>?
fun copy_string(str: CValuesRef<ByteVar /* = ByteVarOf<Byte> */>?, size: Int): Int
```

These declarations look clear. All char * pointers are turned into str: CValuesRef<ByteVar>? for parameters and to CPointer<ByteVar>? in return types. Kotlin turns char type into kotlin.Byte type, as it is usually an 8-bit signed value.

In the generated Kotlin declarations, you see that str is represented as CValuesRef<ByteVar/>?. The type is nullable, and you can simply pass Kotlin null as the parameter value.

Pass Kotlin string to C

Let's try to use the API from Kotlin. Call pass_string first:

```
fun passStringToC() {
  val str = "this is a Kotlin String"
```

```
  pass_string(str.cstr)
}
```

Passing a Kotlin string to C is easy, thanks to the fact that there is String.cstr extension property in Kotlin for it. There is also String.wcstr for cases when you need UTF-16 wide characters.

Read C Strings in Kotlin

This time you'll take a returned char * from the return_string function and turn it into a Kotlin string. For that, do the following in Kotlin:

```
fun passStringToC() {
  val stringFromC = return_string()?.toKString()

  println("Returned from C: $stringFromC")
}
```

This code uses the toKString() extension function above. Please do not miss out the toString() function. The toKString() has two overloaded extension functions in Kotlin:

```
fun CPointer<ByteVar>.toKString(): String
fun CPointer<ShortVar>.toKString(): String
```

The first extension takes a char * as a UTF-8 string and turns it into a String. The second function does the same but for wide UTF-16 strings.

Receive C string bytes from Kotlin

This time we will ask a C function to write us a C string to a given buffer. The function is called copy_string. It takes a pointer to the location writing characters and the allowed buffer size. The function returns something to indicate if it has succeeded or failed. Let's assume 0 means it succeeded, and the supplied buffer was big enough:

```
fun sendString() {
  val buf = ByteArray(255)
  buf.usePinned { pinned ->
    if (copy_string(pinned.addressOf(0), buf.size - 1) != 0) {
      throw Error("Failed to read string from C")
    }
  }

  val copiedStringFromC = buf.decodeToString()
  println("Message from C: $copiedStringFromC")
}
```

First of all, you need to have a native pointer to pass to the C function. Use the usePinned extension function to temporarily pin the native memory address of the byte array. The C function fills in the byte array with data. Use another extension function ByteArray.decodeToString() to turn the byte array into a Kotlin String, assuming UTF-8 encoding.

Fix the Code

You've now seen all the definitions and it is time to fix the code. Run the runDebugExecutableNative Gradle task in the IDE or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

The code in the final hello.kt file may look like this:

```
import interop.*
import kotlinx.cinterop.*

fun main() {
  println("Hello Kotlin/Native!")

  val str = "this is a Kotlin String"
```

```
  pass_string(str.cstr)

  val useMe = return_string()?.toKString() ?: error("null pointer returned")
  println(useMe)

  val copyFromC = ByteArray(255).usePinned { pinned ->

    val useMe2 = copy_string(pinned.addressOf(0), pinned.get().size - 1)
    if (useMe2 != 0) throw Error("Failed to read string from C")
    pinned.get().decodeToString()
  }

  println(copyFromC)
}
```

Next steps

Continue to explore more C language types and their representation in Kotlin/Native in our other tutorials:

- Mapping primitive data types from C
- Mapping struct and union types from C
- Mapping function pointers from C

The C Interop documentation documentation covers more advanced scenarios of the interop.

Create an app using C Interop and libcurl – tutorial

This tutorial demonstrates how to use IntelliJ IDEA to create a command-line application. You'll learn how to create a simple HTTP client that can run natively on specified platforms using Kotlin/Native and the libcurl library.

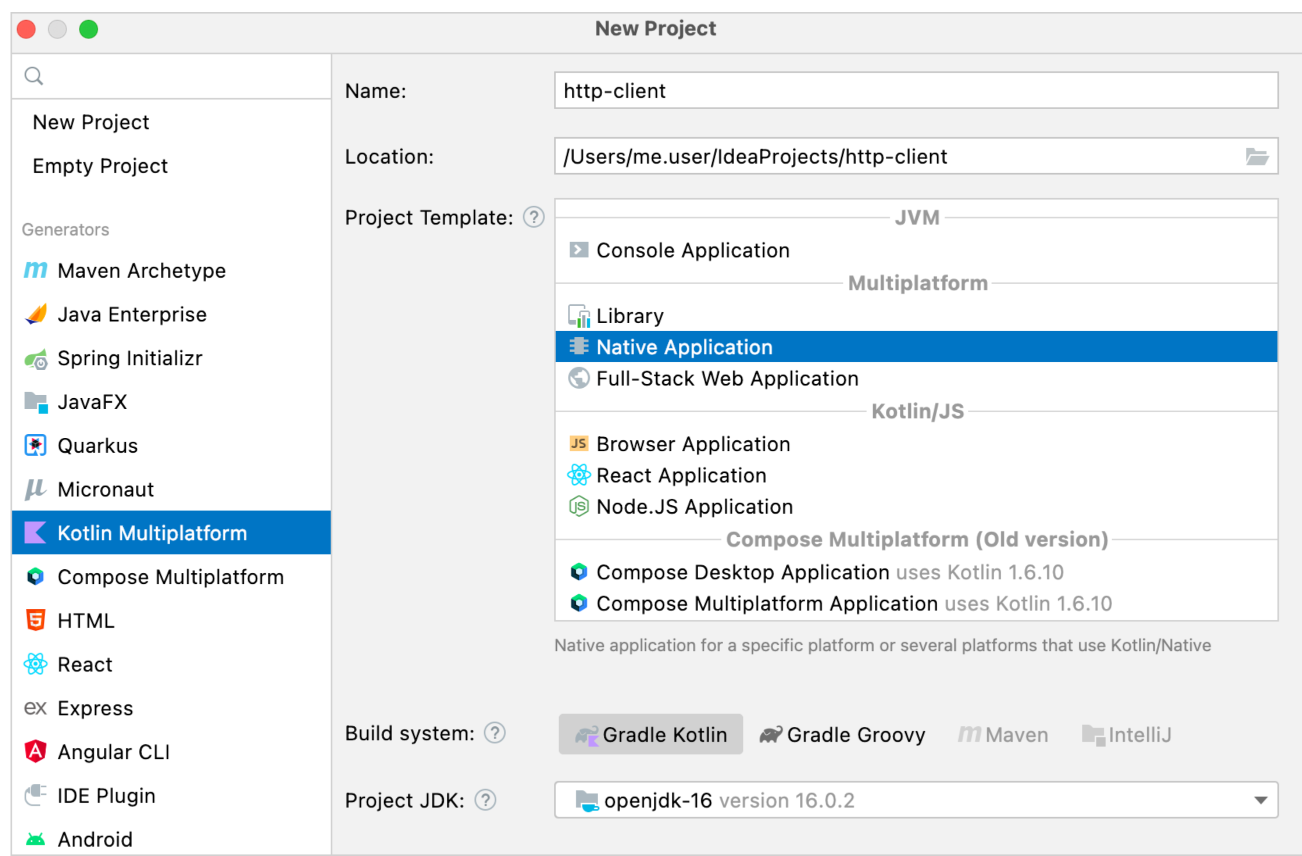
The output will be an executable command-line app that you can run on macOS and Linux and make simple HTTP GET requests.

> While it is possible to use the command line, either directly or by combining it with a script file (such as a .sh or a .bat file), this approach doesn't scale well for big projects with hundreds of files and libraries. In this case, it is better to use the Kotlin/Native compiler with a build system, as it helps download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the Gradle build system through the kotlin-multiplatform Plugin.

To get started, install the latest version of IntelliJ IDEA. The tutorial is suitable for both IntelliJ IDEA Community Edition and IntelliJ IDEA Ultimate.

Create a Kotlin/Native project

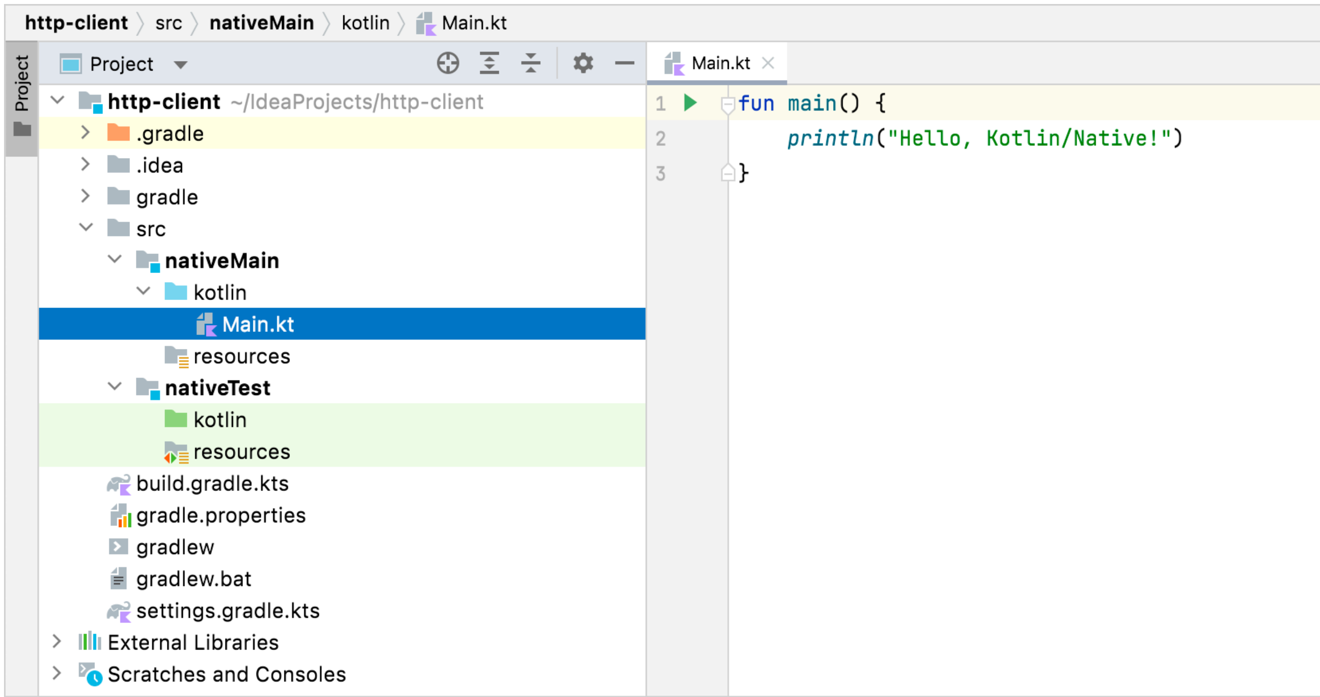
1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select Kotlin Multiplatform | Native Application.
3. Specify the name and select the folder where you'll save your application.



New project. Native application in IntelliJ IDEA

4. Click Next and then Finish.

IntelliJ IDEA will create a new project with the files and folders you need to get you started. It's important to understand that an application written in Kotlin/Native can target different platforms if the code does not have platform-specific requirements. Your code is placed in a folder named NativeMain with its corresponding NativeTest. For this tutorial, keep the folder structure as is.



Native application project structure

Along with your new project, a build.gradle(.kts) file is generated. Pay special attention to the following in the build file:

Kotlin

```kotlin
kotlin {
    val hostOs = System.getProperty("os.name")
    val isMingwX64 = hostOs.startsWith("Windows")
    val nativeTarget = when {
        hostOs == "Mac OS X" -> macosX64("native")
        hostOs == "Linux" -> linuxX64("native")
        isMingwX64 -> mingwX64("native")
        else -> throw GradleException("Host OS is not supported in Kotlin/Native.")
    }

    nativeTarget.apply {
        binaries {
            executable {
                entryPoint = "main"
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    def hostOs = System.getProperty("os.name")
    def isMingwX64 = hostOs.startsWith("Windows")
    def nativeTarget
        if (hostOs == "Mac OS X") nativeTarget = macosX64('native')
        else if (hostOs == "Linux") nativeTarget = linuxX64("native")
        else if (isMingwX64) nativeTarget = mingwX64("native")
        else throw new FileNotFoundException("Host OS is not supported in Kotlin/Native.")

    nativeTarget.with {
        binaries {
            executable {
                entryPoint = 'main'
            }
        }
    }
}
```

- Targets are defined using macOSX64, linuxX64, and mingwX64 for macOS, Linux, and Windows. For a complete list of supported platforms, see the Kotlin Native overview.
- The entry itself defines a series of properties to indicate how the binary is generated and the entry point of the applications. These can be left as default values.
- C interoperability is configured as an additional step in the build. By default, all the symbols from C are imported to the interop package. You may want to import the whole package in .kt files. Learn more about how to configure it.

Create a definition file

When writing native applications, you often need access to certain functionalities that are not included in the Kotlin standard library, such as making HTTP requests, reading and writing from disk, and so on.

Kotlin/Native helps consume standard C libraries, opening up an entire ecosystem of functionality that exists for pretty much anything you may need. Kotlin/Native is already shipped with a set of prebuilt platform libraries, which provide some additional common functionality to the standard library.

An ideal scenario for interop is to call C functions as if you are calling Kotlin functions, following the same signature and conventions. This is when the cinterop tool comes in handy. It takes a C library and generates the corresponding Kotlin bindings, so that the library can be used as if it were Kotlin code.

To generate these bindings, create a library definition .def file that contains some information about the necessary headers. In this app, you'll need the libcurl library to make some HTTP calls. To create a definition file:

1. Select the src folder and create a new directory with File | New | Directory.

2. Name new directory nativeInterop/cinterop. This is the default convention for header file locations, though it can be overridden in the build.gradle(.kts) file if you use a different location.

3. Select this new subfolder and create a new libcurl.def file with File | New | File.

4. Update your file with the following code:

```
headers = curl/curl.h
headerFilter = curl/*

compilerOpts.linux = -I/usr/include -I/usr/include/x86_64-linux-gnu
linkerOpts.osx = -L/opt/local/lib -L/usr/local/opt/curl/lib -lcurl
linkerOpts.linux = -L/usr/lib/x86_64-linux-gnu -lcurl
```

- headers is the list of header files to generate Kotlin stubs. You can add multiple files to this entry, separating each with a \ on a new line. In this case, it's only curl.h. The referenced files need to be available on the system path (in this case, it's /usr/include/curl).

- headerFilter shows what exactly is included. In C, all the headers are also included when one file references another one with the #include directive. Sometimes it's not necessary, and you can add this parameter using glob patterns to fine-tune things.

  headerFilter is an optional argument and is mostly used when the library is installed as a system library. You don't want to fetch external dependencies (such as system stdint.h header) into the interop library. It may be important to optimize the library size and fix potential conflicts between the system and the provided Kotlin/Native compilation environment.

- The next lines are about providing linker and compiler options, which can vary depending on different target platforms. In this case, they are macOS (the .osx suffix) and Linux (the .linux suffix). Parameters without a suffix are also possible (for example, linkerOpts=) and applied to all platforms.

The convention is that each library gets its definition file, usually with the same name as the library. For more information on all the options available to cinterop, see the Interop section.

> You need to have the curl library binaries on your system to make the sample work. On macOS and Linux, it is usually included. On Windows, you can build it from sources (you'll need Visual Studio or Windows SDK Commandline tools). For more details, see the related blog post. Alternatively, you may want to consider a MinGW/MSYS2 curl binary.

Add interoperability to the build process

To use header files, make sure they are generated as a part of the build process. For this, add the following entry to the build.gradle(.kts) file:

Kotlin

```kotlin
nativeTarget.apply {
    compilations.getByName("main") {    // NL
        cinterops {                     // NL
            val libcurl by creating     // NL
        }                               // NL
    }                                   // NL
    binaries {
        executable {
            entryPoint = "main"
        }
    }
}
```

Groovy

```groovy
nativeTarget.with {
    compilations.main { // NL
        cinterops {     // NL
            libcurl     // NL
        }               // NL
    }                   // NL
    binaries {
        executable {
            entryPoint = 'main'
        }
    }
```

```
}
```

The new lines are marked with // NL. First, cinterops is added, and then an entry for each def file. By default, the name of the file is used. You can override this with additional parameters:

Kotlin

```kotlin
val libcurl by creating {
    defFile(project.file("src/nativeInterop/cinterop/libcurl.def"))
    packageName("com.jetbrains.handson.http")
    compilerOpts("-I/path")
    includeDirs.allHeaders("path")
}
```

Groovy

```groovy
libcurl {
    defFile project.file("src/nativeInterop/cinterop/libcurl.def")
    packageName 'com.jetbrains.handson.http'
    compilerOpts '-I/path'
    includeDirs.allHeaders("path")
}
```

See the Interoperability with C section for more details on the available options.

Write the application code

Now you have the library and the corresponding Kotlin stubs and can use them from your application. For this tutorial, convert the simple.c example to Kotlin.

In the src/nativeMain/kotlin/ folder, update your Main.kt file with the following code:

```kotlin
import kotlinx.cinterop.*
import libcurl.*

fun main(args: Array<String>) {
    val curl = curl_easy_init()
    if (curl != null) {
        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com")
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L)
        val res = curl_easy_perform(curl)
        if (res != CURLE_OK) {
            println("curl_easy_perform() failed ${curl_easy_strerror(res)?.toKString()}")
        }
        curl_easy_cleanup(curl)
    }
}
```

As you can see, explicit variable declarations are eliminated in the Kotlin version, but everything else is pretty much the same as the C version. All the calls you'd expect in the libcurl library are available in the Kotlin equivalent.

> This is a line-by-line literal translation. You could also write this in a more Kotlin idiomatic way.

Compile and run the application

1. Compile the application. To do that, call runDebugExecutableNative in the list of run Gradle tasks or use the following command in the terminal:

   ```
   ./gradlew runDebugExecutableNative
   ```

   In this case, the cinterop generated part is implicitly included in the build.

2. If there are no errors during compilation, click the green Run icon in the gutter beside the main() method or use the Alt+Enter shortcut to invoke the launch menu

in IntelliJ IDEA.

IntelliJ IDEA opens the Run tab and shows the output — the contents of https://example.com:

```
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>
```

Application output with HTML-code

You can see the actual output because the call curl_easy_perform prints the result to the standard output. You could hide this using curl_easy_setopt.

> You can get the full code here.

Interoperability with Swift/Objective-C

This document covers some details of Kotlin/Native interoperability with Swift/Objective-C.

Usage

Kotlin/Native provides bidirectional interoperability with Objective-C. Objective-C frameworks and libraries can be used in Kotlin code if properly imported to the build (system frameworks are imported by default). See compilation configurations for more details. A Swift library can be used in Kotlin code if its API is exported to Objective-C with @objc. Pure Swift modules are not yet supported.

Kotlin modules can be used in Swift/Objective-C code if compiled into a framework (see here for how to declare binaries). See Kotlin Multiplatform Mobile Sample for an example.

Hiding Kotlin declarations

If you don't want to export Kotlin declarations to Objective-C and Swift, use special annotations:

- @HiddenFromObjC hides a Kotlin declaration from Objective-C and Swift. The annotation disables a function or property export to Objective-C, making your Kotlin code more Objective-C/Swift-friendly.
- @ShouldRefineInSwift helps to replace a Kotlin declaration with a wrapper written in Swift. The annotation marks a function or property as swift_private in the generated Objective-C API. Such declarations get the __ prefix, which makes them invisible from Swift.

  You can still use these declarations in your Swift code to create a Swift-friendly API, but they won't be suggested in the Xcode autocomplete.

  For more information on refining Objective-C declarations in Swift, see the official Apple documentation.

> Using these annotations requires opt-in.

Mappings

The table below shows how Kotlin concepts are mapped to Swift/Objective-C and vice versa.

"->" and "<-" indicate that mapping only goes one way.

Kotlin	Swift	Objective-C	Notes
class	class	@interface	note
interface	protocol	@protocol	
constructor/create	Initializer	Initializer	note
Property	Property	Property	note 1, note 2
Method	Method	Method	note 1, note 2
suspend->	completionHandler:/async	completionHandler:	note 1, note 2
@Throws fun	throws	error:(NSError**)error	note
Extension	Extension	Category member	note
companion member <-	Class method or property	Class method or property	
null	nil	nil	
Singleton	shared or companion property	shared or companion property	note
Primitive type	Primitive type / NSNumber		note
Unit return type	Void	void	

Kotlin	Swift	Objective-C	Notes
String	String	NSString	
String	NSMutableString	NSMutableString	note
List	Array	NSArray	
MutableList	NSMutableArray	NSMutableArray	
Set	Set	NSSet	
MutableSet	NSMutableSet	NSMutableSet	note
Map	Dictionary	NSDictionary	
MutableMap	NSMutableDictionary	NSMutableDictionary	note
Function type	Function type	Block pointer type	note
Inline classes	Unsupported	Unsupported	note

Name translation

Objective-C classes are imported into Kotlin with their original names. Protocols are imported as interfaces with Protocol name suffix, i.e. @protocol Foo-> interface FooProtocol. These classes and interfaces are placed into a package specified in build configuration (platform.* packages for preconfigured system frameworks).

The names of Kotlin classes and interfaces are prefixed when imported to Objective-C. The prefix is derived from the framework name.

Objective-C does not support packages in a framework. If the Kotlin compiler finds Kotlin classes in the same framework which have the same name but different packages, it renames them. This algorithm is not stable yet and can change between Kotlin releases. To work around this, you can rename the conflicting Kotlin classes in the framework.

To avoid renaming Kotlin declarations, use the @ObjCName annotation. It instructs the Kotlin compiler to use a custom Objective-C and Swift name for classes, interfaces, and other Kotlin concepts:

```
@ObjCName(swiftName = "MySwiftArray")
class MyKotlinArray {
    @ObjCName("index")
    fun indexOf(@ObjCName("of") element: String): Int = TODO()
}

// Usage with the ObjCName annotations
let array = MySwiftArray()
let index = array.index(of: "element")
```

Using this annotation requires opt-in.

Initializers

Swift/Objective-C initializers are imported to Kotlin as constructors and factory methods named create. The latter happens with initializers declared in the Objective-

C category or as a Swift extension, because Kotlin has no concept of extension constructors.

Kotlin constructors are imported as initializers to Swift/Objective-C.

Setters

Writeable Objective-C properties overriding read-only properties of the superclass are represented as setFoo() method for the property foo. The same goes for a protocol's read-only properties that are implemented as mutable.

Top-level functions and properties

Top-level Kotlin functions and properties are accessible as members of special classes. Each Kotlin file is translated into such a class. E.g.

```
// MyLibraryUtils.kt
package my.library

fun foo() {}
```

can be called from Swift like

```
MyLibraryUtilsKt.foo()
```

Method names translation

Generally, Swift argument labels and Objective-C selector pieces are mapped to Kotlin parameter names. These two concepts have different semantics, so sometimes Swift/Objective-C methods can be imported with a clashing Kotlin signature. In this case, the clashing methods can be called from Kotlin using named arguments, e.g.:

```
[player moveTo:LEFT byMeters:17]
[player moveTo:UP byInches:42]
```

In Kotlin, it would be:

```
player.moveTo(LEFT, byMeters = 17)
player.moveTo(UP, byInches = 42)
```

The methods of kotlin.Any (equals(), hashCode() and toString()) are mapped to the methods isEquals:, hash and description in Objective-C, and to the method isEquals(_:) and the properties hash, description in Swift.

You can specify a more idiomatic name in Swift or Objective-C, instead of renaming the Kotlin declaration. Use the @ObjCName annotation that instructs the Kotlin compiler to use a custom Objective-C and Swift name for methods or parameters.

> Using this annotation requires opt-in.

Errors and exceptions

Kotlin has no concept of checked exceptions, all Kotlin exceptions are unchecked. Swift has only checked errors. So if Swift or Objective-C code calls a Kotlin method which throws an exception to be handled, then the Kotlin method should be marked with a @Throws annotation specifying a list of "expected" exception classes.

When compiling to the Objective-C/Swift framework, non-suspend functions that have or inherit the @Throws annotation are represented as NSError*-producing methods in Objective-C and as throws methods in Swift. Representations for suspend functions always have NSError*/Error parameter in completion handler.

When Kotlin function called from Swift/Objective-C code throws an exception which is an instance of one of the @Throws-specified classes or their subclasses, it is propagated as NSError. Other Kotlin exceptions reaching Swift/Objective-C are considered unhandled and cause program termination.

suspend functions without @Throws propagate only CancellationException as NSError. Non-suspend functions without @Throws don't propagate Kotlin exceptions at all.

Note that the opposite reversed translation is not implemented yet: Swift/Objective-C error-throwing methods aren't imported to Kotlin as exception-throwing.

Suspending functions

> Support for calling suspend functions from Swift code as async is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin's suspending functions (suspend) are presented in the generated Objective-C headers as functions with callbacks, or completion handlers in Swift/Objective-C terminology.

Starting from Swift 5.5, Kotlin's suspend functions are also available for calling from Swift as async functions without using the completion handlers. Currently, this functionality is highly experimental and has certain limitations. See this YouTrack issue for details.

Learn more about the async/await mechanism in Swift.

Extensions and category members

Members of Objective-C categories and Swift extensions are generally imported to Kotlin as extensions. That's why these declarations can't be overridden in Kotlin. And the extension initializers aren't available as Kotlin constructors.

> Currently, there are two exceptions. Starting with Kotlin 1.8.20, category members that are declared in the same headers as the NSView class (from the AppKit framework) or UIView classes (from the UIKit framework) are imported as members of these classes. This means that you can override methods that subclass from NSView or UIView.

Kotlin extensions to "regular" Kotlin classes are imported to Swift and Objective-C as extensions and category members, respectively. Kotlin extensions to other types are treated as top-level declarations with an additional receiver parameter. These types include:

- Kotlin String type
- Kotlin collection types and subtypes
- Kotlin interface types
- Kotlin primitive types
- Kotlin inline classes
- Kotlin Any type
- Kotlin function types and subtypes
- Objective-C classes and protocols

Kotlin singletons

Kotlin singleton (made with an object declaration, including companion object) is imported to Swift/Objective-C as a class with a single instance.

The instance is available through the shared and companion properties.

For the following Kotlin code:

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

Access these objects as follows:

```
MyObject.shared
MyObject.shared.x
MyClass.companion
```

```
MyClass.Companion.shared
```

> Access objects through [MySingleton mySingleton] in Objective-C and MySingleton() in Swift has been deprecated.

NSNumber

Kotlin primitive type boxes are mapped to special Swift/Objective-C classes. For example, kotlin.Int box is represented as KotlinInt class instance in Swift (or ${prefix}Int instance in Objective-C, where prefix is the framework names prefix). These classes are derived from NSNumber, so the instances are proper NSNumbers supporting all corresponding operations.

NSNumber type is not automatically translated to Kotlin primitive types when used as a Swift/Objective-C parameter type or return value. The reason is that NSNumber type doesn't provide enough information about a wrapped primitive value type, i.e. NSNumber is statically not known to be Byte, Boolean, or Double. So Kotlin primitive values should be cast to/from NSNumber manually (see below).

NSMutableString

NSMutableString Objective-C class is not available from Kotlin. All instances of NSMutableString are copied when passed to Kotlin.

Collections

Kotlin collections are converted to Swift/Objective-C collections as described in the table above. Swift/Objective-C collections are mapped to Kotlin in the same way, except for NSMutableSet and NSMutableDictionary. NSMutableSet isn't converted to a Kotlin MutableSet. To pass an object for Kotlin MutableSet, you can create this kind of Kotlin collection explicitly by either creating it in Kotlin with e.g. mutableSetOf(), or using the KotlinMutableSet class in Swift (or ${prefix}MutableSet in Objective-C, where prefix is the framework names prefix). The same holds for MutableMap.

Function types

Kotlin function-typed objects (e.g. lambdas) are converted to Swift functions / Objective-C blocks. However, there is a difference in how types of parameters and return values are mapped when translating a function and a function type. In the latter case, primitive types are mapped to their boxed representation. Kotlin Unit return value is represented as a corresponding Unit singleton in Swift/Objective-C. The value of this singleton can be retrieved in the same way as it is for any other Kotlin object (see singletons in the table above). To sum the things up:

```
fun foo(block: (Int) -> Unit) { ... }
```

would be represented in Swift as

```
func foo(block: (KotlinInt) -> KotlinUnit)
```

and can be called like

```
foo {
    bar($0 as! Int32)
    return KotlinUnit()
}
```

Generics

Objective-C supports "lightweight generics" defined on classes, with a relatively limited feature set. Swift can import generics defined on classes to help provide additional type information to the compiler.

Generic feature support for Objective-C and Swift differ from Kotlin, so the translation will inevitably lose some information, but the features supported retain meaningful information.

Limitations

Objective-C generics do not support all features of either Kotlin or Swift, so there will be some information lost in the translation.

Generics can only be defined on classes, not on interfaces (protocols in Objective-C and Swift) or functions.

Nullability

Kotlin and Swift both define nullability as part of the type specification, while Objective-C defines nullability on methods and properties of a type. As such, the following:

```
class Sample<T>() {
    fun myVal(): T
}
```

will (logically) look like this:

```
class Sample<T>() {
    fun myVal(): T?
}
```

In order to support a potentially nullable type, the Objective-C header needs to define myVal with a nullable return value.

To mitigate this, when defining your generic classes, if the generic type should never be null, provide a non-null type constraint:

```
class Sample<T : Any>() {
    fun myVal(): T
}
```

That will force the Objective-C header to mark myVal as non-null.

Variance

Objective-C allows generics to be declared covariant or contravariant. Swift has no support for variance. Generic classes coming from Objective-C can be force-cast as needed.

```
data class SomeData(val num: Int = 42) : BaseData()
class GenVarOut<out T : Any>(val arg: T)
```

```
let variOut = GenVarOut<SomeData>(arg: sd)
let variOutAny : GenVarOut<BaseData> = variOut as! GenVarOut<BaseData>
```

Constraints

In Kotlin, you can provide upper bounds for a generic type. Objective-C also supports this, but that support is unavailable in more complex cases, and is currently not supported in the Kotlin - Objective-C interop. The exception here being a non-null upper bound will make Objective-C methods/properties non-null.

To disable

To have the framework header written without generics, add the flag to the compiler config:

```
binaries.framework {
    freeCompilerArgs += "-Xno-objc-generics"
}
```

Casting between mapped types

When writing Kotlin code, an object may need to be converted from a Kotlin type to the equivalent Swift/Objective-C type (or vice versa). In this case, a plain old Kotlin cast can be used, e.g.

```
val nsArray = listOf(1, 2, 3) as NSArray
val string = nsString as String
val nsNumber = 42 as NSNumber
```

Subclassing

Subclassing Kotlin classes and interfaces from Swift/Objective-C

Kotlin classes and interfaces can be subclassed by Swift/Objective-C classes and protocols.

Subclassing Swift/Objective-C classes and protocols from Kotlin

Swift/Objective-C classes and protocols can be subclassed with a Kotlin final class. Non-final Kotlin classes inheriting Swift/Objective-C types aren't supported yet, so it is not possible to declare a complex class hierarchy inheriting Swift/Objective-C types.

Normal methods can be overridden using the override Kotlin keyword. In this case, the overriding method must have the same parameter names as the overridden one.

Sometimes it is required to override initializers, e.g. when subclassing UIViewController. Initializers imported as Kotlin constructors can be overridden by Kotlin constructors marked with the @OverrideInit annotation:

```
class ViewController : UIViewController {
    @OverrideInit constructor(coder: NSCoder) : super(coder)

    ...
}
```

The overriding constructor must have the same parameter names and types as the overridden one.

To override different methods with clashing Kotlin signatures, you can add a @Suppress("CONFLICTING_OVERLOADS") annotation to the class.

By default, the Kotlin/Native compiler doesn't allow calling a non-designated Objective-C initializer as a super(...) constructor. This behaviour can be inconvenient if the designated initializers aren't marked properly in the Objective-C library. Adding a disableDesignatedInitializerChecks = true to the .def file for this library would disable these compiler checks.

C features

See Interoperability with C for an example case where the library uses some plain C features, such as unsafe pointers, structs, and so on.

Export of KDoc comments to generated Objective-C headers

> The ability to export KDoc comments to generated Objective-C headers is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

By default, KDocs documentation comments are not translated into corresponding comments when generating an Objective-C header.
For example, the following Kotlin code with KDoc:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
fun printSum(a: Int, b: Int) = println(a.toLong() + b)
```

will produce an Objective-C declaration without any comments:

```
+ (void)printSumA:(int32_t)a b:(int32_t)b __attribute__((swift_name("printSum(a:b:)")));
```

To enable export of KDoc comments, add the following compiler option to your build.gradle(.kts):

Kotlin

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        compilations.get("main").compilerOptions.options.freeCompilerArgs.add("-Xexport-kdoc")
    }
}
```

Groovy

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        compilations.get("main").compilerOptions.options.freeCompilerArgs.add("-Xexport-kdoc")
    }
}
```

After that, the Objective-C header will contain a corresponding comment:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
+ (void)printSumA:(int32_t)a b:(int32_t)b __attribute__((swift_name("printSum(a:b:)")));
```

Known limitations:

- Dependency documentation is not exported unless it is compiled with -Xexport-kdoc itself. The feature is experimental, so libraries compiled with this flag might be incompatible with other compiler versions.
- KDoc comments are mostly exported "as is". Many KDoc features (for example, @property) are not supported.

Unsupported

Some features of Kotlin programming language are not yet mapped into respective features of Objective-C or Swift. Currently, the following features are not properly exposed in generated framework headers:

- inline classes (arguments are mapped as either underlying primitive type or id)
- custom classes implementing standard Kotlin collection interfaces (List, Map, Set) and other special classes
- Kotlin subclasses of Objective-C classes

Kotlin/Native as an Apple framework – tutorial

Kotlin/Native provides bi-directional interoperability with Objective-C/Swift. Objective-C frameworks and libraries can be used in Kotlin code. Kotlin modules can be used in Swift/Objective-C code too. Besides that, Kotlin/Native has C Interop. There is also the Kotlin/Native as a Dynamic Library tutorial for more information.

In this tutorial, you will see how to use Kotlin/Native code from Objective-C and Swift applications on macOS and iOS.

In this tutorial you'll:

- create a Kotlin Library and compile it to a framework
- examine the generated Objective-C and Swift API code
- use the framework from Objective-C and Swift
- Configure Xcode to use the framework for macOS and iOS

Create a Kotlin library

The Kotlin/Native compiler can produce a framework for macOS and iOS out of the Kotlin code. The created framework contains all declarations and binaries needed to use it with Objective-C and Swift. The best way to understand the techniques is to try it for ourselves. Let's create a tiny Kotlin library first and use it from an Objective-C program.

Create the hello.kt file with the library contents:

```
package example

object Object {
  val field = "A"
```

```
}

interface Interface {
  fun iMember() {}
}

class Clazz : Interface {
  fun member(p: Int): ULong? = 42UL
}

fun forIntegers(b: Byte, s: UShort, i: Int, l: ULong?) { }
fun forFloats(f: Float, d: Double?) { }

fun strings(str: String?) : String {
  return "That is '$str' from C"
}

fun acceptFun(f: (String) -> String?) = f("Kotlin/Native rocks!")
fun supplyFun() : (String) -> String? = { "$it is cool!" }
```

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is therefore better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the Gradle build system through the kotlin-multiplatform plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the A Basic Kotlin/Native Application tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and multiplatform builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}

repositories {
    mavenCentral()
}

kotlin {
  macosX64("native") {
    binaries {
      framework {
        baseName = "Demo"
      }
    }
  }
}

tasks.wrapper {
  gradleVersion = "7.6"
  distributionType = Wrapper.DistributionType.ALL
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}

repositories {
    mavenCentral()
}

kotlin {
  macosX64("native") {
    binaries {
      framework {
        baseName = "Demo"
```

```
      }
    }
  }
}

wrapper {
  gradleVersion = "7.6"
  distributionType = "ALL"
}
```

Move the sources file into the src/nativeMain/kotlin folder under the project. That is the default path, where sources are located, when the kotlin-multiplatform plugin is used. Use the following block to configure the project to generate a dynamic or shared library:

```
binaries {
  framework {
    baseName = "Demo"
  }
}
```

Along with macOS X64, Kotlin/Native supports macos arm64 and iOS arm32, arm64 and X64 targets. You may replace the macosX64 with respective functions as shown in the table:

Target platform/device	Gradle function
macOS x86_64	macosX64()
macOS ARM 64	macosArm64()
iOS ARM 64	iosArm64()
iOS Simulator (x86_64)	iosX64()
iOS Simulator (arm64)	iosSimulatorArm64

Run the linkNative Gradle task to build the library in the IDE or by calling the following console command:

```
./gradlew linkNative
```

Depending on the variant, the build generates the framework into the build/bin/native/debugFramework and build/bin/native/releaseFramework folders. Let's see what is inside.

Generated framework headers

Each of the created frameworks contains the header file in <Framework>/Headers/Demo.h. The headers do not depend on the target platform (at least with Kotlin/Native v.0.9.2). It contains the definitions for our Kotlin code and a few Kotlin-wide declarations.

> The way Kotlin/Native exports symbols is subject to change without notice.

Kotlin/Native runtime declarations

Take a look at Kotlin runtime declarations:

```
NS_ASSUME_NONNULL_BEGIN

@interface KotlinBase : NSObject
```

```
- (instancetype)init __attribute__((unavailable));
+ (instancetype)new __attribute__((unavailable));
+ (void)initialize __attribute__((objc_requires_super));
@end;

@interface KotlinBase (KotlinBaseCopying) <NSCopying>
@end;

__attribute__((objc_runtime_name("KotlinMutableSet")))
__attribute__((swift_name("KotlinMutableSet")))
@interface DemoMutableSet<ObjectType> : NSMutableSet<ObjectType>
@end;

__attribute__((objc_runtime_name("KotlinMutableDictionary")))
__attribute__((swift_name("KotlinMutableDictionary")))
@interface DemoMutableDictionary<KeyType, ObjectType> : NSMutableDictionary<KeyType, ObjectType>
@end;

@interface NSError (NSErrorKotlinException)
@property (readonly) id _Nullable kotlinException;
@end;
```

Kotlin classes have a KotlinBase base class in Objective-C, the class extends the NSObject class there. There are also have wrappers for collections and exceptions. Most of the collection types are mapped to similar collection types from the other side:

Kotlin	Swift	Objective-C
List	Array	NSArray
MutableList	NSMutableArray	NSMutableArray
Set	Set	NSSet
Map	Dictionary	NSDictionary
MutableMap	NSMutableDictionary	NSMutableDictionary

Kotlin numbers and NSNumber

The next part of the <Framework>/Headers/Demo.h contains number type mappings between Kotlin/Native and NSNumber. There is the base class called DemoNumber in Objective-C and KotlinNumber in Swift. It extends NSNumber. There are also child classes per Kotlin number type:

Kotlin	Swift	Objective-C	Simple type
-	KotlinNumber	<Package>Number	-
Byte	KotlinByte	<Package>Byte	char
UByte	KotlinUByte	<Package>UByte	unsigned char
Short	KotlinShort	<Package>Short	short
UShort	KotlinUShort	<Package>UShort	unsigned short

Kotlin	Swift	Objective-C	Simple type
Int	KotlinInt	<Package>Int	int
UInt	KotlinUInt	<Package>UInt	unsigned int
Long	KotlinLong	<Package>Long	long long
ULong	KotlinULong	<Package>ULong	unsigned long long
Float	KotlinFloat	<Package>Float	float
Double	KotlinDouble	<Package>Double	double
Boolean	KotlinBoolean	<Package>Boolean	BOOL/Bool

Every number type has a class method to create a new instance from the related simple type. Also, there is an instance method to extract a simple value back. Schematically, declarations look like that:

```
__attribute__((objc_runtime_name("Kotlin__TYPE__")))
__attribute__((swift_name("Kotlin__TYPE__")))
@interface Demo__TYPE__ : DemoNumber
- (instancetype)initWith__TYPE__:(__CTYPE__)value;
+ (instancetype)numberWith__TYPE__:(__CTYPE__)value;
@end;
```

Where __TYPE__ is one of the simple type names and __CTYPE__ is the related Objective-C type, for example, initWithChar(char).

These types are used to map boxed Kotlin number types into Objective-C and Swift. In Swift, you may simply call the constructor to create an instance, for example, KotlinLong(value: 42).

Classes and objects from Kotlin

Let's see how class and object are mapped to Objective-C and Swift. The generated <Framework>/Headers/Demo.h file contains the exact definitions for Class, Interface, and Object:

```
NS_ASSUME_NONNULL_BEGIN

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Object")))
@interface DemoObject : KotlinBase
+ (instancetype)alloc __attribute__((unavailable));
+ (instancetype)allocWithZone:(struct _NSZone *)zone __attribute__((unavailable));
+ (instancetype)object __attribute__((swift_name("init()")));
@property (readonly) NSString *field;
@end;

__attribute__((swift_name("Interface")))
@protocol DemoInterface
@required
- (void)iMember __attribute__((swift_name("iMember()")));
@end;

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Clazz")))
@interface DemoClazz : KotlinBase <DemoInterface>
- (instancetype)init __attribute__((swift_name("init()"))) __attribute__((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable, message="use object initializers instead")));
- (DemoLong * _Nullable)memberP:(int32_t)p __attribute__((swift_name("member(p:)")));
@end;
```

The code is full of Objective-C attributes, which are intended to help the use of the framework from both Objective-C and Swift languages. DemoClazz, DemoInterface, and DemoObject are created for Clazz, Interface, and Object respectively. The Interface is turned into @protocol, both a class and an object are represented as @interface. The Demo prefix comes from the -output parameter of the kotlinc-native compiler and the framework name. You can see here that the nullable return type ULong? is turned into DemoLong* in Objective-C.

Global declarations from Kotlin

All global functions from Kotlin are turned into DemoLibKt in Objective-C and into LibKt in Swift, where Demo is the framework name and set by the -output parameter of kotlinc-native.

```
NS_ASSUME_NONNULL_BEGIN

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("LibKt")))
@interface DemoLibKt : KotlinBase
+ (void)forIntegersB:(int8_t)b s:(int16_t)s i:(int32_t)i l:(DemoLong * _Nullable)l
__attribute__((swift_name("forIntegers(b:s:i:l:)")));
+ (void)forFloatsF:(float)f d:(DemoDouble * _Nullable)d __attribute__((swift_name("forFloats(f:d:)")));
+ (NSString *)stringsStr:(NSString * _Nullable)str __attribute__((swift_name("strings(str:)")));
+ (NSString * _Nullable)acceptFunF:(NSString * _Nullable (^)(NSString *))f __attribute__((swift_name("acceptFun(f:)")));
+ (NSString * _Nullable (^)(NSString *))supplyFun __attribute__((swift_name("supplyFun()")));
@end;
```

You see that Kotlin String and Objective-C NSString* are mapped transparently. Similarly, Unit type from Kotlin is mapped to void. We see primitive types are mapped directly. Non-nullable primitive types are mapped transparently. Nullable primitive types are mapped into Kotlin<TYPE>* types, as shown in the table above. Both higher order functions acceptFunF and supplyFun are included, and accept Objective-C blocks.

More information about all other types mapping details can be found in the Objective-C Interop documentation article

Garbage collection and reference counting

Objective-C and Swift use reference counting. Kotlin/Native has its own garbage collection too. Kotlin/Native garbage collection is integrated with Objective-C/Swift reference counting. You do not need to use anything special to control the lifetime of Kotlin/Native instances from Swift or Objective-C.

Use the code from Objective-C

Let's call the framework from Objective-C. For that, create the main.m file with the following content:

```
#import <Foundation/Foundation.h>
#import <Demo/Demo.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        [[DemoObject object] field];

        DemoClazz* clazz = [[ DemoClazz alloc] init];
        [clazz memberP:42];

        [DemoLibKt forIntegersB:1 s:1 i:3 l:[DemoULong numberWithUnsignedLongLong:4]];
        [DemoLibKt forIntegersB:1 s:1 i:3 l:nil];

        [DemoLibKt forFloatsF:2.71 d:[DemoDouble numberWithDouble:2.71]];
        [DemoLibKt forFloatsF:2.71 d:nil];

        NSString* ret = [DemoLibKt acceptFunF:^NSString * _Nullable(NSString * it) {
            return [it stringByAppendingString:@" Kotlin is fun"];
        }];

        NSLog(@"%@", ret);
        return 0;
    }
}
```

Here you call Kotlin classes directly from Objective-C code. A Kotlin object has the class method function object, which allows us to get the only instance of the object and to call Object methods on it. The widespread pattern is used to create an instance of the Clazz class. You call the [[DemoClazz alloc] init] on Objective-C. You may also use [DemoClazz new] for constructors without parameters. Global declarations from the Kotlin sources are scoped under the DemoLibKt class in Objective-C. All methods are turned into class methods of that class. The strings function is turned into DemoLibKt.stringsStr function in Objective-C, you can pass

NSString directly to it. The return is visible as NSString too.

Use the code from Swift

The framework that you compiled with Kotlin/Native has helper attributes to make it easier to use with Swift. Convert the previous Objective-C example into Swift. As a result, you'll have the following code in main.swift:

```
import Foundation
import Demo

let kotlinObject = Object()
assert(kotlinObject === Object(), "Kotlin object has only one instance")

let field = Object().field

let clazz = Clazz()
clazz.member(p: 42)

LibKt.forIntegers(b: 1, s: 2, i: 3, l: 4)
LibKt.forFloats(f: 2.71, d: nil)

let ret = LibKt.acceptFun { "\($0) Kotlin is fun" }
if (ret != nil) {
  print(ret!)
}
```

The Kotlin code is turned into very similar looking code in Swift. There are some small differences, though. In Kotlin any object has only one instance. Kotlin object Object now has a constructor in Swift, and we use the Object() syntax to access the only instance of it. The instance is always the same in Swift, so that Object() === Object() is true. Methods and property names are translated as-is. Kotlin String is turned into Swift String too. Swift hides NSNumber* boxing from us too. We can pass a Swift closure to Kotlin and call a Kotlin lambda function from Swift too.

More documentation on the types mapping can be found in the Objective-C Interop article.

Xcode and framework dependencies

You need to configure an Xcode project to use our framework. The configuration depends on the target platform.

Xcode for macOS target

First, in the General tab of the target configuration, under the Linked Frameworks and Libraries section, you need to include our framework. This will make Xcode look at our framework and resolve imports both from Objective-C and Swift.

The second step is to configure the framework search path of the produced binary. It is also known as rpath or run-time search path. The binary uses the path to look for the required frameworks. We do not recommend installing additional frameworks to the OS if it is not needed. You should understand the layout of your future application, for example, you may have the Frameworks folder under the application bundle with all the frameworks you use. The @rpath parameter can be configured in Xcode. You need to open the project configuration and find the Runpath Search Paths section. Here you specify the relative path to the compiled framework.

Xcode for iOS targets

First, you need to include the compiled framework in the Xcode project. To do this, add the framework to the Frameworks, Libraries, and Embedded Content section of the General tab of the target configuration page.

The second step is to then include the framework path into the Framework Search Paths section of the Build Settings tab of the target configuration page. It is possible to use the $(PROJECT_DIR) macro to simplify the setup.

The iOS simulator requires a framework compiled for the ios_x64 target, the iOS_sim folder in our case.

This Stackoverflow thread contains a few more recommendations. Also, the CocoaPods package manager may be helpful to automate the process too.

Next steps

Kotlin/Native has bidirectional interop with Objective-C and Swift languages. Kotlin objects integrate with Objective-C/Swift reference counting. Unused Kotlin objects are automatically removed. The Objective-C Interop article contains more information on the interop implementation details. Of course, it is possible to

import an existing framework and use it from Kotlin. Kotlin/Native comes with a good set of pre-imported system frameworks.

Kotlin/Native supports C interop too. Check out the Kotlin/Native as a Dynamic Library tutorial for that.

CocoaPods overview and setup

Kotlin/Native provides integration with the CocoaPods dependency manager. You can add dependencies on Pod libraries as well as use a multiplatform project with native targets as a CocoaPods dependency.

You can manage Pod dependencies directly in IntelliJ IDEA and enjoy all the additional features such as code highlighting and completion. You can build the whole Kotlin project with Gradle and not ever have to switch to Xcode.

Use Xcode only when you need to write Swift/Objective-C code or run your application on a simulator or device. To work correctly with Xcode, you should update your Podfile.

Depending on your project and purposes, you can add dependencies between a Kotlin project and a Pod library as well as a Kotlin Gradle project and an Xcode project.

Set up an environment to work with CocoaPods

Install the CocoaPods dependency manager using the installation tool of your choice:

RVM

1. Install Ruby version manager in case you don't have yet.
2. Install Ruby. You can choose a specific version:

```
rvm install ruby 3.0.0
```

3. Install CocoaPods:

```
sudo gem install -n /usr/local/bin cocoapods
```

Rbenv

1. Install rbenv from GitHub in case you don't have yet.
2. Install Ruby. You can choose a specific version:

```
rbenv install 3.0.0
```

3. Set the Ruby version as local for a particular directory or global for the whole machine:

```
rbenv global 3.0.0
```

4. Install CocoaPods:

```
sudo gem install cocoapods
```

Default Ruby

> This way of installation doesn't work on devices with Apple M chips. Use other tools to set up an environment to work with CocoaPods.

You can install the CocoaPods dependency manager with the default Ruby that should be available on macOS:

```
sudo gem install cocoapods
```

Homebrew

The CocoaPods installation with Homebrew might result in compatibility issues.

When installing CocoaPods, Homebrew also installs the Xcodeproj gem that is necessary for working with Xcode. However, it cannot be updated with Homebrew, and if the installed Xcodeproj doesn't support the newest Xcode version yet, you'll get errors with Pod installation. If this is the case, try other tools to install CocoaPods.

1. Install Homebrew in case you don't have yet.
2. Install CocoaPods:

```
brew install cocoapods
```

If you use Kotlin prior to version 1.7.0
If your current version of Kotlin is earlier than 1.7.0, additionally install the cocoapods-generate plugin:

```
sudo gem install -n /usr/local/bin cocoapods-generate
```

Mind that cocoapods-generate couldn't be installed on Ruby 3.0.0 and later. If it's your case, downgrade Ruby or upgrade Kotlin to 1.7.0 or later.

If you encounter problems during the installation, check the Possible issues and solutions section.

Add and configure Kotlin CocoaPods Gradle plugin

If your environment is set up correctly, you can create a new Kotlin Multiplatform project and choose CocoaPods Dependency Manager as the iOS framework distribution option. The plugin will automatically generate the project for you.

If you want to configure your project manually:

1. In build.gradle(.kts) of your project, apply the CocoaPods plugin as well as the Kotlin Multiplatform plugin:

```
plugins {
    kotlin("multiplatform") version "1.9.0"
    kotlin("native.cocoapods") version "1.9.0"
}
```

2. Configure version, summary, homepage, and baseName of the Podspec file in the cocoapods block:

```
plugins {
    kotlin("multiplatform") version "1.9.0"
    kotlin("native.cocoapods") version "1.9.0"
}

kotlin {
    cocoapods {
        // Required properties
        // Specify the required Pod version here. Otherwise, the Gradle project version is used.
        version = "1.0"
        summary = "Some description for a Kotlin/Native module"
        homepage = "Link to a Kotlin/Native module homepage"

        // Optional properties
        // Configure the Pod name here instead of changing the Gradle project name
        name = "MyCocoaPod"

        framework {
            // Required properties
            // Framework name configuration. Use this property instead of deprecated 'frameworkName'
            baseName = "MyFramework"

            // Optional properties
            // Specify the framework linking type. It's dynamic by default.
```

```
            isStatic = false
            // Dependency export
            export(project(":anotherKMMModule"))
            transitiveExport = false // This is default.
            // Bitcode embedding
            embedBitcode(BITCODE)
        }

        // Maps custom Xcode configuration to NativeBuildType
        xcodeConfigurationToNativeBuildType["CUSTOM_DEBUG"] = NativeBuildType.DEBUG
        xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] = NativeBuildType.RELEASE
    }
}
```

> See the full syntax of Kotlin DSL in the Kotlin Gradle plugin repository.

3. Re-import the project.
4. Generate the Gradle wrapper to avoid compatibility issues during an Xcode build.

When applied, the CocoaPods plugin does the following:

- Adds both debug and release frameworks as output binaries for all macOS, iOS, tvOS, and watchOS targets.
- Creates a podspec task which generates a Podspec file for the project.

The Podspec file includes a path to an output framework and script phases that automate building this framework during the build process of an Xcode project.

Update Podfile for Xcode

If you want to import your Kotlin project in an Xcode project, you need to make some changes to your Podfile:

- If your project has any Git, HTTP, or custom Podspec repository dependencies, you should also specify the path to the Podspec in the Podfile.

  For example, if you add a dependency on podspecWithFilesExample, declare the path to the Podspec in the Podfile:

  ```
  target 'ios-app' do
      # ... other dependencies ...
      pod 'podspecWithFilesExample', :path => 'cocoapods/externalSources/url/podspecWithFilesExample'
  end
  ```

  The :path should contain the filepath to the Pod.

- When you add a library from the custom Podspec repository, you should also specify the location of specs at the beginning of your Podfile:

  ```
  source 'https://github.com/Kotlin/kotlin-cocoapods-spec.git'

  target 'kotlin-cocoapods-xcproj' do
      # ... other dependencies ...
      pod 'example'
  end
  ```

> Re-import the project after making changes in the Podfile.

If you don't make these changes to the Podfile, the podInstall task will fail, and the CocoaPods plugin will show an error message in the log.

Check out the withXcproject branch of the sample project, which contains an example of Xcode integration with an existing Xcode project named kotlin-cocoapods-xcproj.

Possible issues and solutions

CocoaPods installation

Ruby installation

CocoaPods is built with Ruby, and you can install it with the default Ruby that should be available on macOS. Ruby 1.9 or later has a built-in RubyGems package management framework that helps you install the CocoaPods dependency manager.

If you're experiencing problems installing CocoaPods and getting it to work, follow this guide to install Ruby or refer to the RubyGems website to install the framework.

Version compatibility

We recommend using the latest Kotlin version. If your current version is earlier than 1.7.0, you'll need to additionally install the cocoapods-generate plugin.

However, cocoapods-generate is not compatible with Ruby 3.0.0 or later. In this case, downgrade Ruby or upgrade Kotlin to 1.7.0 or later.

Module not found

You may encounter a module 'SomeSDK' not found error that is connected with the C-interop issue. Try these workarounds to avoid this error:

Specify the framework name

1. Look through the downloaded Pod directory [shared_module_name]/build/cocoapods/synthetic/IOS/Pods/... for the module.modulemap file.

2. Check the framework name inside the module, for example AppsFlyerLib {}. If the framework name doesn't match the Pod name, specify it explicitly:

```
pod("AFNetworking") {
    moduleName = "AppsFlyerLib"
}
```

Check the definition file

If the Pod doesn't contain a .modulemap file, like the pod("NearbyMessages"), in the generated .def file, replace modules with headers with the pointing main header:

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.DefFileTask>("generateDefNearbyMessages").configure {
    doLast {
        outputFile.writeText("""
            language = Objective-C
            headers = GNSMessages.h
        """.trimIndent())
    }
}
```

Check the CocoaPods documentation for more information. If nothing works, and you still encounter this error, report an issue in YouTrack.

Add dependencies on a Pod library

To add dependencies between a Kotlin project and a Pod library, complete the initial configuration. You can then add dependencies on different types of Pod libraries.

When you add a new dependency and re-import the project in IntelliJ IDEA, the new dependency will be added automatically. No additional steps are required.

To use your Kotlin project with Xcode, you should make changes in your project Podfile.

A Kotlin project requires the pod() function call in build.gradle(.kts) for adding a Pod dependency. Each dependency requires its separate function call. You can specify the parameters for the dependency in the configuration block of the function.

> If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

You can find a sample project here.

From the CocoaPods repository

1. Specify the name of a Pod library in the pod() function.

   In the configuration block, you can specify the version of the library using the version parameter. To use the latest version of the library, you can just omit this parameter altogether.

   > You can add dependencies on subspecs.

2. Specify the minimum deployment target version for the Pod library.

```
kotlin {
    ios()

    cocoapods {
        ios.deploymentTarget = "13.5"

        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        pod("AFNetworking") {
            version = "~> 4.0.1"
        }
    }
}
```

3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages cocoapods.<library-name>:

```
import cocoapods.AFNetworking.*
```

On a locally stored library

1. Specify the name of a Pod library in the pod() function.

   In the configuration block, specify the path to the local Pod library: use the path() function in the source parameter value.

   > You can add local dependencies on subspecs as well. The cocoapods block can include dependencies to Pods stored locally and Pods from the CocoaPods repository at the same time.

2. Specify the minimum deployment target version for the Pod library.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("pod_dependency") {
            version = "1.0"
            source = path(project.file("../pod_dependency"))
        }
        pod("subspec_dependency/Core") {
            version = "1.0"
            source = path(project.file("../subspec_dependency"))
        }
        pod("AFNetworking") {
            version = "~> 4.0.1"
        }
    }
}
```

> You can also specify the version of the library using version parameter in the configuration block. To use the latest version of the library, omit the parameter.

3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages cocoapods.<library-name>:

```
import cocoapods.pod_dependency.*
import cocoapods.subspec_dependency.*
import cocoapods.AFNetworking.*
```

From a custom Git repository

1. Specify the name of a Pod library in the pod() function.

   In the configuration block, specify the path to the git repository: use the git() function in the source parameter value.

   Additionally, you can specify the following parameters in the block after git():

   - commit – to use a specific commit from the repository
   - tag – to use a specific tag from the repository
   - branch – to use a specific branch from the repository

   The git() function prioritizes passed parameters in the following order: commit, tag, branch. If you don't specify a parameter, the Kotlin plugin uses HEAD from the master branch.

   > You can combine branch, commit, and tag parameters to get the specific version of a Pod.

2. Specify the minimum deployment target version for the Pod library.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("AFNetworking") {
            source = git("https://github.com/AFNetworking/AFNetworking") {
                tag = "4.0.0"
            }
        }

        pod("JSONModel") {
            source = git("https://github.com/jsonmodel/jsonmodel.git") {
                branch = "key-mapper-class"
            }
        }

        pod("CocoaLumberjack") {
            source = git("https://github.com/CocoaLumberjack/CocoaLumberjack.git") {
                commit = "3e7f595e3a459c39b917aacf9856cd2a48c4dbf3"
            }
        }
    }
}
```

3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages cocoapods.<library-name>:

```
import cocoapods.AFNetworking.*
import cocoapods.JSONModel.*
```

```
import cocoapods.CocoaLumberjack.*
```

From a custom Podspec repository

1. Specify the HTTP address to the custom Podspec repository using the url() inside the specRepos block.
2. Specify the name of a Pod library in the pod() function.
3. Specify the minimum deployment target version for the Pod library.

   ```
   kotlin {
       ios()

       cocoapods {
           summary = "CocoaPods test library"
           homepage = "https://github.com/JetBrains/kotlin"

           ios.deploymentTarget = "13.5"

           specRepos {
               url("https://github.com/Kotlin/kotlin-cocoapods-spec.git")
           }
           pod("example")
       }
   }
   ```

4. Re-import the project.

> To work correctly with Xcode, you should specify the location of specs at the beginning of your Podfile. For example,
>
> ```
> source 'https://github.com/Kotlin/kotlin-cocoapods-spec.git'
> ```

To use these dependencies from the Kotlin code, import the packages cocoapods.<library-name>:

```
import cocoapods.example.*
```

With custom cinterop options

1. Specify the name of a Pod library in the pod() function.

   In the configuration block, specify the cinterop options:

   - extraOpts – to specify the list of options for a Pod library. For example, specific flags: extraOpts = listOf("-compiler-option").
   - packageName – to specify the package name. If you specify this, you can import the library using the package name: import <packageName>.

2. Specify the minimum deployment target version for the Pod library.

   ```
   kotlin {
       ios()

       cocoapods {
           summary = "CocoaPods test library"
           homepage = "https://github.com/JetBrains/kotlin"

           ios.deploymentTarget = "13.5"

           pod("YandexMapKit") {
               packageName = "YandexMK"
           }
       }
   }
   ```

3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages cocoapods.<library-name>:

```
import cocoapods.YandexMapKit.*
```

If you use the packageName parameter, you can import the library using the package name import <packageName>:

```
import YandexMK.YMKPoint
import YandexMK.YMKDistance
```

Support for Objective-C headers with @import directives

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We'd appreciate your feedback on it in YouTrack.

Some Objective-C libraries, specifically those that serve as wrappers for Swift libraries, have @import directives in their headers. By default, cinterop doesn't provide support for these directives.

To enable support for @import directives, specify the -fmodules option in the configuration block of the pod() function:

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("PodName") {
            extraOpts = listOf("-compiler-option", "-fmodules")
        }
    }
}
```

Use a Kotlin Gradle project as a CocoaPods dependency

To use a Kotlin Multiplatform project with native targets as a CocoaPods dependency, complete the initial configuration. You can include such a dependency in the Podfile of the Xcode project by its name and path to the project directory containing the generated Podspec.

This dependency will be automatically built (and rebuilt) along with this project. Such an approach simplifies importing to Xcode by removing a need to write the corresponding Gradle tasks and Xcode build steps manually.

You can add dependencies between a Kotlin Gradle project and an Xcode project with one or several targets. It's also possible to add dependencies between a Gradle project and multiple Xcode projects. However, in this case, you need to add a dependency by calling pod install manually for each Xcode project. In other cases, it's done automatically.

> - To correctly import the dependencies into the Kotlin/Native module, the Podfile must contain either use_modular_headers! or use_frameworks! directive.
> - If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

Xcode project with one target

1. Create an Xcode project with a Podfile if you haven't done so yet.

2. Add the path to your Xcode project Podfile with podfile = project.file(..) to build.gradle(.kts) of your Kotlin project. This step helps synchronize your Xcode project with Gradle project dependencies by calling pod install for your Podfile.

3. Specify the minimum deployment target version for the Pod library.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"
        ios.deploymentTarget = "13.5"
        pod("AFNetworking") {
            version = "~> 4.0.0"
        }
        podfile = project.file("../ios-app/Podfile")
    }
}
```

4. Add the name and path of the Gradle project you want to include in the Xcode project to Podfile.

```
use_frameworks!

platform :ios, '13.5'

target 'ios-app' do
        pod 'kotlin_library', :path => '../kotlin-library'
end
```

5. Re-import the project.

Xcode project with several targets

1. Create an Xcode project with a Podfile if you haven't done so yet.

2. Add the path to your Xcode project Podfile with podfile = project.file(..) to build.gradle(.kts) of your Kotlin project. This step helps synchronize your Xcode project with Gradle project dependencies by calling pod install for your Podfile.

3. Add dependencies to the Pod libraries you want to use in your project with pod().

4. For each target, specify the minimum deployment target version for the Pod library.

```
kotlin {
    ios()
    tvos()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"
        ios.deploymentTarget = "13.5"
        tvos.deploymentTarget = "13.4"

        pod("AFNetworking") {
            version = "~> 4.0.0"
        }
        podfile = project.file("../severalTargetsXcodeProject/Podfile") // specify the path to the Podfile
    }
}
```

5. Add the name and path of the Gradle project you want to include in the Xcode project to the Podfile.

```
target 'iosApp' do
  use_frameworks!
  platform :ios, '13.5'
  # Pods for iosApp
  pod 'kotlin_library', :path => '../kotlin-library'
end

target 'TVosApp' do
  use_frameworks!
  platform :tvos, '13.4'
```

```
  # Pods for TVosApp
  pod 'kotlin_library', :path => '../kotlin-library'
end
```

6. Re-import the project.

You can find a sample project here.

CocoaPods Gradle plugin DSL reference

Kotlin CocoaPods Gradle plugin is a tool for creating Podspec files. These files are necessary to integrate your Kotlin project with the CocoaPods dependency manager.

This reference contains the complete list of blocks, functions, and properties for the Kotlin CocoaPods Gradle plugin that you can use when working with the CocoaPods integration.

- Learn how to set up the environment and configure the Kotlin CocoaPods Gradle plugin.
- Depending on your project and purposes, you can add dependencies between a Kotlin project and a Pod library as well as a Kotlin Gradle project and an Xcode project.

Enable the plugin

To apply the CocoaPods plugin, add the following lines to the build.gradle(.kts) file:

```
plugins {
    kotlin("multiplatform") version "1.9.0"
    kotlin("native.cocoapods") version "1.9.0"
}
```

The plugin versions match the Kotlin release versions. The latest stable version is 1.9.0.

cocoapods block

The cocoapods block is the top-level block for the CocoaPods configuration. It contains general information on the Pod, including required information like the Pod version, summary, and homepage, as well as optional features.

You can use the following blocks, functions, and properties inside it:

Name	Description
version	The version of the Pod. If this is not specified, a Gradle project version is used. If none of these properties are configured, you'll get an error.
summary	A required description of the Pod built from this project.
homepage	A required link to the homepage of the Pod built from this project.
authors	Specifies authors of the Pod built from this project.
podfile	Configures the existing Podfile file.

Name	Description
noPodspec()	Sets up the plugin not to produce a Podspec file for the cocoapods section.
name	The name of the Pod built from this project. If not provided, the project name is used.
license	The license of the Pod built from this project, its type, and the text.
framework	The framework block configures the framework produced by the plugin.
source	The location of the Pod built from this project.
extraSpecAttributes	Configures other Podspec attributes like libraries or vendored_frameworks.
xcodeConfigurationToNativeBuildType	Maps custom Xcode configuration to NativeBuildType: "Debug" to NativeBuildType.DEBUG and "Release" to NativeBuildType.RELEASE.
publishDir	Configures the output directory for Pod publishing.
pods	Returns a list of Pod dependencies.
pod()	Adds a CocoaPods dependency to the Pod built from this project.
specRepos	Adds a specification repository using url(). This is necessary when a private Pod is used as a dependency. See the CocoaPods documentation for more information.

Targets

- ios
- osx
- tvos
- watchos

For each target, use the deploymentTarget property to specify the minimum target version for the Pod library.

When applied, CocoaPods adds both debug and release frameworks as output binaries for all of the targets.

```
kotlin {
    ios()

    cocoapods {
        version = "2.0"
        name = "MyCocoaPod"
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        extraSpecAttributes["vendored_frameworks"] = 'CustomFramework.xcframework'
        license = "{ :type => 'MIT', :text => 'License text'}"
        source = "{ :git => 'git@github.com:vkormushkin/kmmpodlibrary.git', :tag => '$version' }"
        authors = "Kotlin Dev"
```

```
        specRepos {
            url("https://github.com/Kotlin/kotlin-cocoapods-spec.git")
        }
        pod("example")

        xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] = NativeBuildType.RELEASE
    }
}
```

framework block

The framework block is nested inside cocoapods and configures the framework properties of the Pod built from the project.

> Note that baseName is a required field.

Name	Description
baseName	A required framework name. Use this property instead of the deprecated frameworkName.
isStatic	Defines the framework linking type. It's dynamic by default.
transitiveExport	Enables dependency export.

```
kotlin {
    cocoapods {
        framework {
            baseName = "MyFramework"
            isStatic = false
            export(project(":anotherKMMModule"))
            transitiveExport = true
        }
    }
}
```

pod() function

The pod() function call adds a CocoaPods dependency to the Pod built from this project. Each dependency requires a separate function call.

You can specify the name of a Pod library in the function parameters and additional parameter values, like the version and source of the library, in its configuration block:

Name	Description
version	The library version. To use the latest version of the library, omit the parameter.
source	Configures the Pod from: • The Git repository using git(). In the block after git(), you can specify commit to use a specific commit, tag to use a specific tag, and branch to use a specific branch from the repository • The local repository using path() • An archived (tar, jar, zip) Pod folder using url()

Name	Description
packageName	Specifies the package name.
extraOpts	Specifies the list of options for a Pod library. For example, specific flags: `extraOpts = listOf("-compiler-option")`
linkOnly	Instructs the CocoaPods plugin to use pod dependencies with dynamic frameworks without generating C-interop bindings. If used with static frameworks, the option will remove the Pod dependency entirely.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("pod_dependency") {
            version = "1.0"
            linkOnly = true
            source = path(project.file("../pod_dependency"))
        }
    }
}
```

Kotlin/Native libraries

Kotlin compiler specifics

To produce a library with the Kotlin/Native compiler use the -produce library or -p library flag. For example:

```
$ kotlinc-native foo.kt -p library -o bar
```

This command will produce a bar.klib with the compiled contents of foo.kt.

To link to a library use the -library <name> or -l <name> flag. For example:

```
$ kotlinc-native qux.kt -l bar
```

This command will produce a program.kexe out of qux.kt and bar.klib

cinterop tool specifics

The cinterop tool produces .klib wrappers for native libraries as its main output. For example, using the simple libgit2.def native library definition file provided in your Kotlin/Native distribution

```
$ cinterop -def samples/gitchurn/src/nativeInterop/cinterop/libgit2.def -compiler-option -I/usr/local/include -o libgit2
```

we will obtain libgit2.klib.

See more details in C Interop.

klib utility

The klib library management utility allows you to inspect and install the libraries.

The following commands are available:

- content – list library contents:

```
$ klib contents <name>
```

- info – inspect the bookkeeping details of the library

```
$ klib info <name>
```

- install – install the library to the default location use

```
$ klib install <name>
```

- remove – remove the library from the default repository use

```
$ klib remove <name>
```

All of the above commands accept an additional -repository <directory> argument for specifying a repository different to the default one.

```
$ klib <command> <name> -repository <directory>
```

Several examples

First let's create a library. Place the tiny library source code into kotlinizer.kt:

```
package kotlinizer
val String.kotlinized
    get() = "Kotlin $this"
```

```
$ kotlinc-native kotlinizer.kt -p library -o kotlinizer
```

The library has been created in the current directory:

```
$ ls kotlinizer.klib
kotlinizer.klib
```

Now let's check out the contents of the library:

```
$ klib contents kotlinizer
```

You can install kotlinizer to the default repository:

```
$ klib install kotlinizer
```

Remove any traces of it from the current directory:

```
$ rm kotlinizer.klib
```

Create a very short program and place it into a use.kt:

```
import kotlinizer.*

fun main(args: Array<String>) {
```

```
    println("Hello, ${"world".kotlinized}!")
}
```

Now compile the program linking with the library you have just created:

```
$ kotlinc-native use.kt -l kotlinizer -o kohello
```

And run the program:

```
$ ./kohello.kexe
Hello, Kotlin world!
```

Have fun!

Advanced topics

Library search sequence

When given a -library foo flag, the compiler searches the foo library in the following order:

- Current compilation directory or an absolute path.
- All repositories specified with -repo flag.
- Libraries installed in the default repository (For now the default is ~/.konan, however it could be changed by setting KONAN_DATA_DIR environment variable).
- Libraries installed in $installation/klib directory.

Library format

Kotlin/Native libraries are zip files containing a predefined directory structure, with the following layout:

foo.klib when unpacked as foo/ gives us:

```
  - foo/
    - $component_name/
      - ir/
        - Serialized Kotlin IR.
      - targets/
        - $platform/
          - kotlin/
            - Kotlin compiled to LLVM bitcode.
          - native/
            - Bitcode files of additional native objects.
        - $another_platform/
          - There can be several platform specific kotlin and native pairs.
      - linkdata/
        - A set of ProtoBuf files with serialized linkage metadata.
      - resources/
        - General resources such as images. (Not used yet).
      - manifest - A file in the java property format describing the library.
```

An example layout can be found in klib/stdlib directory of your installation.

Using relative paths in klibs

Using relative paths in klibs is available since Kotlin 1.6.20.

A serialized IR representation of source files is a part of a klib library. It includes paths of files for generating proper debug information. By default, stored paths are absolute. With the -Xklib-relative-path-base compiler option, you can change the format and use only relative path in the artifact. To make it work, pass one or multiple base paths of source files as an argument:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
    // $base is a base path of source files
    compilerOptions.freeCompilerArgs.add("-Xklib-relative-path-base=$base")
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        // $base is a base path of source files
        freeCompilerArgs.add("-Xklib-relative-path-base=$base")
    }
}
```

Platform libraries

To provide access to user's native operating system services, Kotlin/Native distribution includes a set of prebuilt libraries specific to each target. We call them Platform Libraries.

POSIX bindings

For all Unix- or Windows-based targets (including Android and iOS targets) we provide the POSIX platform lib. It contains bindings to platform's implementation of the POSIX standard.

To use the library, just import it:

```kotlin
import platform.posix.*
```

The only target for which it is not available is WebAssembly.

Note that the content of platform.posix is NOT identical on different platforms, in the same way as different POSIX implementations are a little different.

Popular native libraries

There are many more platform libraries available for host and cross-compilation targets. Kotlin/Native distribution provides access to OpenGL, zlib and other popular native libraries on applicable platforms.

On Apple platforms, objc library is provided for interoperability with Objective-C.

Inspect the contents of dist/klib/platform/$target of the distribution for the details.

Availability by default

The packages from platform libraries are available by default. No special link flags need to be specified to use them. Kotlin/Native compiler automatically detects which of the platform libraries have been accessed and automatically links the needed libraries.

On the other hand, the platform libs in the distribution are merely just wrappers and bindings to the native libraries. That means the native libraries themselves (.so, .a, .dylib, .dll etc) should be installed on the machine.

Kotlin/Native as a dynamic library – tutorial

Learn how you can use the Kotlin/Native code from existing native applications or libraries. For this, you need to compile the Kotlin code into a dynamic library, .so, .dylib, and .dll.

Kotlin/Native also has tight integration with Apple technologies. The Kotlin/Native as an Apple Framework tutorial explains how to compile Kotlin code into a framework for Swift and Objective-C.

In this tutorial, you will:

- Compile Kotlin code to a dynamic library
- Examine generated C headers
- Use the Kotlin dynamic library from C
- Compile and run the example on Linux and Mac and Windows

Create a Kotlin library

Kotlin/Native compiler can produce a dynamic library out of the Kotlin code. A dynamic library often comes with a header file, a .h file, which you will use to call the compiled code from C.

The best way to understand these techniques is to try them out. First, create a first tiny Kotlin library and use it from a C program.

Start by creating a library file in Kotlin and save it as hello.kt:

```
package example

object Object {
  val field = "A"
}

class Clazz {
  fun memberFunction(p: Int): ULong = 42UL
}

fun forIntegers(b: Byte, s: Short, i: UInt, l: Long) { }
fun forFloats(f: Float, d: Double) { }

fun strings(str: String) : String? {
  return "That is '$str' from C"
}

val globalString = "A global String"
```

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the Gradle build system through the kotlin-multiplatform plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the A Basic Kotlin/Native Application tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and multiplatform builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64("native") { // on Linux
```

```
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64("native") { // on Windows
    binaries {
      sharedLib {
        baseName = "native" // on Linux and macOS
        // baseName = "libnative" // on Windows
      }
    }
  }
}

tasks.wrapper {
  gradleVersion = "7.6"
  distributionType = Wrapper.DistributionType.ALL
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}

repositories {
    mavenCentral()
}

kotlin {
  linuxX64("native") { // on Linux
  // macosX64("native") { // on x86_64 macOS
  // macosArm64("native") { // on Apple Silicon macOS
  // mingwX64("native") { // on Windows
    binaries {
      sharedLib {
        baseName = "native" // on Linux and macOS
        // baseName = "libnative" // on Windows
      }
    }
  }
}

wrapper {
  gradleVersion = "7.6"
  distributionType = "ALL"
}
```

Move the sources file into the src/nativeMain/kotlin folder under the project. This is the default path, for where sources are located, when the kotlin-multiplatform plugin is used. Use the following block to instruct and configure the project to generate a dynamic or shared library:

```
binaries {
  sharedLib {
    baseName = "native" // on Linux and macOS
    // baseName = "libnative" // on Windows
  }
}
```

The libnative is used as the library name, the generated header file name prefix. It is also prefixes all declarations in the header file.

Now you can open the project in IntelliJ IDEA and to see how to fix the example project. While doing this, we'll examine how C functions are mapped into Kotlin/Native declarations.

Run the linkNative Gradle task to build the library in the IDE or by calling the following console command:

```
./gradlew linkNative
```

The build generates the following files under the build/bin/native/debugShared folder, depending on the host OS:

- macOS: libnative_api.h and libnative.dylib
- Linux: libnative_api.h and libnative.so
- Windows: libnative_api.h, libnative_symbols.def and libnative.dll

The same rules are used by the Kotlin/Native compiler to generate the .h file for all platforms.

Let's check out the C API of our Kotlin library.

Generated headers file

In the libnative_api.h, you'll find the following code. Let's discuss the code in parts to make it easier to understand.

> The way Kotlin/Native exports symbols is subject to change without notice.

The very first part contains the standard C/C++ header and footer:

```
#ifndef KONAN_DEMO_H
#define KONAN_DEMO_H
#ifdef __cplusplus
extern "C" {
#endif

/// THE REST OF THE GENERATED CODE GOES HERE

#ifdef __cplusplus
}  /* extern "C" */
#endif
#endif  /* KONAN_DEMO_H */
```

After the rituals in the libnative_api.h, there is a block with the common type definitions:

```
#ifdef __cplusplus
typedef bool            libnative_KBoolean;
#else
typedef _Bool           libnative_KBoolean;
#endif
typedef unsigned short     libnative_KChar;
typedef signed char        libnative_KByte;
typedef short              libnative_KShort;
typedef int                libnative_KInt;
typedef long long          libnative_KLong;
typedef unsigned char      libnative_KUByte;
typedef unsigned short     libnative_KUShort;
typedef unsigned int       libnative_KUInt;
typedef unsigned long long libnative_KULong;
typedef float              libnative_KFloat;
typedef double             libnative_KDouble;
typedef void*              libnative_KNativePtr;
```

Kotlin uses the libnative_ prefix for all declarations in the created libnative_api.h file. Let's present the mapping of the types in a more readable way:

Kotlin Define	C Type
libnative_KBoolean	bool or _Bool
libnative_KChar	unsigned short
libnative_KByte	signed char
libnative_KShort	short
libnative_KInt	int

Kotlin Define	C Type
libnative_KLong	long long
libnative_KUByte	unsigned char
libnative_KUShort	unsigned short
libnative_KUInt	unsigned int
libnative_KULong	unsigned long long
libnative_KFloat	float
libnative_KDouble	double
libnative_KNativePtr	void*

The definitions part shows how Kotlin primitive types map into C primitive types. The reverse mapping is described in the Mapping primitive data types from C tutorial.

The next part of the libnative_api.h file contains definitions of the types that are used in the library:

```
struct libnative_KType;
typedef struct libnative_KType libnative_KType;

typedef struct {
  libnative_KNativePtr pinned;
} libnative_kref_example_Object;

typedef struct {
  libnative_KNativePtr pinned;
} libnative_kref_example_Clazz;
```

The typedef struct { .. } TYPE_NAME syntax is used in C language to declare a structure. This thread on Stackoverflow provides more explanations of that pattern.

As you can see from these definitions, the Kotlin object Object is mapped into libnative_kref_example_Object, and Clazz is mapped into libnative_kref_example_Clazz. Both structs contain nothing but the pinned field with a pointer, the field type libnative_KNativePtr is defined as void* above.

There is no namespaces support in C, so the Kotlin/Native compiler generates long names to avoid any possible clashes with other symbols in the existing native project.

A significant part of the definitions goes in the libnative_api.h file. It includes the definition of our Kotlin/Native library world:

```
typedef struct {
  /* Service functions. */
  void (*DisposeStablePointer)(libnative_KNativePtr ptr);
  void (*DisposeString)(const char* string);
  libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* type);

  /* User functions. */
  struct {
    struct {
      struct {
        void (*forIntegers)(libnative_KByte b, libnative_KShort s, libnative_KUInt i, libnative_KLong l);
        void (*forFloats)(libnative_KFloat f, libnative_KDouble d);
        const char* (*strings)(const char* str);
        const char* (*get_globalString)();
        struct {
```

```
          libnative_KType* (*_type)(void);
          libnative_kref_example_Object (*_instance)();
          const char* (*get_field)(libnative_kref_example_Object thiz);
        } Object;
        struct {
          libnative_KType* (*_type)(void);
          libnative_kref_example_Clazz (*Clazz)();
          libnative_KULong (*memberFunction)(libnative_kref_example_Clazz thiz, libnative_KInt p);
        } Clazz;
      } example;
    } root;
  } kotlin;
} libnative_ExportedSymbols;
```

The code uses anonymous structure declarations. The code struct { .. } foo declares a field in the outer struct of that anonymous structure type, the type with no name.

C does not support objects either. People use function pointers to mimic object semantics. A function pointer is declared as follows RETURN_TYPE (* FIELD_NAME)(PARAMETERS). It is tricky to read, but we should be able to see function pointer fields in the structures above.

Runtime functions

The code reads as follows. You have the libnative_ExportedSymbols structure, which defines all the functions that Kotlin/Native and our library provides us. It uses nested anonymous structures heavily to mimic packages. The libnative_ prefix comes from the library name.

The libnative_ExportedSymbols structure contains several helper functions:

```
void (*DisposeStablePointer)(libnative_KNativePtr ptr);
void (*DisposeString)(const char* string);
libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* type);
```

These functions deal with Kotlin/Native objects. Call the DisposeStablePointer to release a Kotlin object and DisposeString to release a Kotlin String, which has the char* type in C. It is possible to use the IsInstance function to check if a Kotlin type or a libnative_KNativePtr is an instance of another type. The actual set of operations generated depends on the actual usages.

Kotlin/Native has garbage collection, but it does not help us deal with Kotlin objects from the C language. Kotlin/Native has interop with Objective-C and Swift and integrates with their reference counters. The Objective-C Interop documentation article contains more details on it. Also, there is the tutorial Kotlin/Native as an Apple Framework.

Your library functions

Let's take a look at the kotlin.root.example field, it mimics the package structure of our Kotlin code with a kotlin.root. prefix.

There is a kotlin.root.example.Clazz field that represents the Clazz from Kotlin. The Clazz#memberFunction is accessible with the memberFunction field. The only difference is that the memberFunction accepts a this reference as the first parameter. The C language does not support objects, and this is the reason to pass a this pointer explicitly.

There is a constructor in the Clazz field (aka kotlin.root.example.Clazz.Clazz), which is the constructor function to create an instance of the Clazz.

Kotlin object Object is accessible as kotlin.root.example.Object. There is the _instance function to get the only instance of the object.

Properties are translated into functions. The get_ and set_ prefix is used to name the getter and the setter functions respectively. For example, the read-only property globalString from Kotlin is turned into a get_globalString function in C.

Global functions forInts, forFloats, or strings are turned into the functions pointers in the kotlin.root.example anonymous struct.

Entry point

You can see how the API is created. To start with, you need to initialize the libnative_ExportedSymbols structure. Let's take a look at the latest part of the libnative_api.h for this:

```
extern libnative_ExportedSymbols* libnative_symbols(void);
```

The function libnative_symbols allows you to open the way from the native code to the Kotlin/Native library. This is the entry point you'll use. The library name is used as a prefix for the function name.

Kotlin/Native object references do not support multi-threaded access. Hosting the returned libnative_ExportedSymbols* pointer per thread might be necessary.

Use generated headers from C

The usage from C is straightforward and uncomplicated. Create a main.c file with the following code:

```
#include "libnative_api.h"
#include "stdio.h"

int main(int argc, char** argv) {
  //obtain reference for calling Kotlin/Native functions
  libnative_ExportedSymbols* lib = libnative_symbols();

  lib->kotlin.root.example.forIntegers(1, 2, 3, 4);
  lib->kotlin.root.example.forFloats(1.0f, 2.0);

  //use C and Kotlin/Native strings
  const char* str = "Hello from Native!";
  const char* response = lib->kotlin.root.example.strings(str);
  printf("in: %s\nout:%s\n", str, response);
  lib->DisposeString(response);

  //create Kotlin object instance
  libnative_kref_example_Clazz newInstance = lib->kotlin.root.example.Clazz.Clazz();
  long x = lib->kotlin.root.example.Clazz.memberFunction(newInstance, 42);
  lib->DisposeStablePointer(newInstance.pinned);

  printf("DemoClazz returned %ld\n", x);

  return 0;
}
```

Compile and run the example on Linux and macOS

On macOS 10.13 with Xcode, compile the C code and link it with the dynamic library with the following command:

```
clang main.c libnative.dylib
```

On Linux call a similar command:

```
gcc main.c libnative.so
```

The compiler generates an executable called a.out. Run it to see in action the Kotlin code being executed from C library. On Linux, you'll need to include . into the LD_LIBRARY_PATH to let the application know to load the libnative.so library from the current folder.

Compile and run the example on Windows

To start with, you'll need a Microsoft Visual C++ compiler installed that supports a x64_64 target. The easiest way to do this is to have a version of Microsoft Visual Studio installed on a Windows machine.

In this example, you'll be using the x64 Native Tools Command Prompt <VERSION> console. You'll see the shortcut to open the console in the start menu. It comes with a Microsoft Visual Studio package.

On Windows, Dynamic libraries are included either via a generated static library wrapper or with manual code, which deals with the LoadLibrary or similar Win32API functions. Follow the first option and generate the static wrapper library for the libnative.dll as described below.

Call lib.exe from the toolchain to generate the static library wrapper libnative.lib that automates the DLL usage from the code:

```
lib /def:libnative_symbols.def /out:libnative.lib
```

Now you are ready to compile our main.c into an executable. Include the generated libnative.lib into the build command and start:

```
cl.exe main.c libnative.lib
```

The command produces the main.exe file, which you can run.

Next steps

Dynamic libraries are the main way to use Kotlin code from existing programs. You can use them to share your code with many platforms or languages, including JVM, Python, iOS, Android, and others.

Kotlin/Native also has tight integration with Objective-C and Swift. It is covered in the Kotlin/Native as an Apple Framework tutorial.

Kotlin/Native memory management

> This page describes features of the new memory manager enabled by default since Kotlin 1.7.20. See our migration guide to move your projects from the legacy memory manager.

Kotlin/Native uses a modern memory manager that is similar to JVM, Go, and other mainstream technologies:

- Objects are stored in a shared heap and can be accessed from any thread.
- Tracing garbage collector (GC) is executed periodically to collect objects that are not reachable from the "roots", like local and global variables.

The memory manager is the same across all the Kotlin/Native targets, except for wasm32, which is only supported in the legacy memory manager.

Garbage collector

The exact algorithm of GC is constantly evolving. As of 1.7.20, it is the Stop-the-World Mark and Concurrent Sweep collector that does not separate heap into generations.

GC is executed on a separate thread and kicked off based on the timer and memory pressure heuristics, or can be called manually.

Enable garbage collection manually
To force start garbage collector, call kotlin.native.internal.GC.collect(). It triggers a new collection and waits for its completion.

Monitor GC performance
There are no special instruments to monitor the GC performance yet. However, it's still possible to look through GC logs for diagnosis. To enable logging, set the following compilation flag in the Gradle build script:

```
-Xruntime-logs=gc=info
```

Currently, the logs are only printed to stderr.

Disable garbage collection
It's recommended to keep GC enabled. However, you can disable it in certain cases, for example, for testing purposes or if you encounter issues and have a short-lived program. To do that, set the following compilation flag in the Gradle build script:

```
-Xgc=noop
```

> With this option enabled, GC doesn't collect Kotlin objects, so memory consumption will keep rising as long as the program runs. Be careful not to exhaust the system memory.

Memory consumption

With the Kotlin/Native memory manager, it's possible to monitor memory consumption. You can check for memory leaks and adjust memory consumption if necessary.

Check for memory leaks

To access the memory manager metrics, call kotlin.native.internal.GC.lastGCInfo(). It returns statistics for the last run of the garbage collector. The statistics can be useful for:

- Debugging memory leaks when using global variables
- Checking if there are any leaks when running tests

```
import kotlin.native.internal.*
import kotlin.test.*

class Resource

val global = mutableListOf<Resource>()

@OptIn(ExperimentalStdlibApi::class)
fun getUsage() : Long {
    GC.collect()
    return GC.lastGCInfo!!.memoryUsageAfter["heap"]!!.totalObjectsSizeBytes
}

fun run() {
    global.add(Resource())
    // The test will fail if you remove the next line
    global.clear()
}

@Test
fun test() {
    val before = getUsage()
    // A separate function is used to ensure that all temporary objects are cleared
    run()
    val after = getUsage()
    assertEquals(before, after)
}
```

Adjust memory consumption

If there are no memory leaks in the program, but you still see unexpectedly high memory consumption, try updating Kotlin to the latest version. We're constantly improving the memory manager, so even a simple compiler update might improve memory consumption.

Another way to fix high memory consumption is related to mimalloc, the default memory allocator for many targets. It pre-allocates and holds onto the system memory to improve the allocation speed.

To avoid that at the cost of performance, a couple of options are available:

- Switch the memory allocator from mimalloc to the system allocator. For that, set the -Xallocator=std compilation option in your Gradle build script.
- Since Kotlin 1.8.0-Beta, you can also instruct mimalloc to promptly release memory back to the system. It's a smaller performance cost, but it gives less definitive results.

  For that, enable the following binary option in your gradle.properties file:

  ```
  kotlin.native.binary.mimallocUseCompaction=true
  ```

If none of these options improved the memory consumption, report an issue in YouTrack.

Unit tests in the background

In unit tests, nothing processes the main thread queue, so don't use Dispatchers.Main unless it was mocked, which can be done by calling Dispatchers.setMain from kotlinx-coroutines-test.

If you don't rely on kotlinx.coroutines or Dispatchers.setMain doesn't work for you for some reason, try the following workaround for implementing the test launcher:

package testlauncher import platform.CoreFoundation.* import kotlin.native.concurrent.* import kotlin.native.internal.test.* import kotlin.system.* fun mainBackground(args: Array<String>) { val worker = Worker.start(name = "main-background") worker.execute(TransferMode.SAFE, { args.freeze() }) { val result = testLauncherEntryPoint(it) exitProcess(result) } CFRunLoopRun() error("CFRunLoopRun should never return") }

Then, compile the test binary with the -e testlauncher.mainBackground compiler flag.

Legacy memory manager

If it's necessary, you can switch back to the legacy memory manager. Set the following option in your gradle.properties:

```
kotlin.native.binary.memoryModel=strict
```

- Compiler cache support is not available for the legacy memory manager, so compilation times might become worse.
- This Gradle option for reverting to the legacy memory manager will be removed in future releases.

If you encounter issues with migrating from the legacy memory manager, or you want to temporarily support both the current and legacy memory managers, see our recommendations in the migration guide.

What's next

- Migrate from the legacy memory manager
- Configure integration with iOS

iOS integration

This page describes the new memory manager enabled by default since Kotlin 1.7.20. See our migration guide to move your projects from the legacy memory manager.

Integration of Kotlin/Native garbage collector with Swift/Objective-C ARC is seamless and generally requires no additional work to be done. Learn more about Swift/Objective-C interoperability.

However, there are some specifics you should keep in mind:

Threads

Deinitializers

Deinit on the Swift/Objective-C objects and the objects they refer to is called on a different thread if these objects cross interop boundaries into Kotlin/Native, for example:

```
// Kotlin
class KotlinExample {
    fun action(arg: Any) {
        println(arg)
    }
}
```

```
// Swift
class SwiftExample {
    init() {
        print("init on \(Thread.current)")
    }
```

```
    deinit {
        print("deinit on \(Thread.current)")
    }
}

func test() {
    KotlinExample().action(arg: SwiftExample())
}
```

The resulting output:

```
init on <_NSMainThread: 0x600003bc0000>{number = 1, name = main}
shared.SwiftExample
deinit on <NSThread: 0x600003b9b900>{number = 7, name = (null)}
```

Completion handlers

When calling Kotlin suspending functions from Swift, completion handlers might be called on threads other than the main one, for example:

```
// Kotlin
// coroutineScope, launch, and delay are from kotlinx.coroutines
suspend fun asyncFunctionExample() = coroutineScope {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

```
// Swift
func test() {
    print("Running test on \(Thread.current)")
    PlatformKt.asyncFunctionExample(completionHandler: { _ in
        print("Running completion handler on \(Thread.current)")
    })
}
```

The resulting output:

```
Running test on <_NSMainThread: 0x600001b100c0>{number = 1, name = main}
Hello
World!
Running completion handler on <NSThread: 0x600001b45bc0>{number = 7, name = (null)}
```

Calling Kotlin suspending functions

The Kotlin/Native memory manager has a restriction on calling Kotlin suspending functions from Swift and Objective-C from threads other than the main one.

This restriction was originally introduced in the legacy memory manager due to cases when the code dispatched a continuation to be resumed on the original thread. If this thread didn't have a supported event loop, the task would never run, and the coroutine would never be resumed.

In certain cases, this restriction is not required anymore. You can lift it by adding the following option to your gradle.properties:

```
kotlin.native.binary.objcExportSuspendFunctionLaunchThreadRestriction=none
```

Garbage collection and lifecycle

Object reclamation

An object is reclaimed only during the garbage collection. This applies to Swift/Objective-C objects that cross interop boundaries into Kotlin/Native, for example:

```
// Kotlin
class KotlinExample {
    fun action(arg: Any) {
```

```
        println(arg)
    }
}
```

```
// Swift
class SwiftExample {
    deinit {
        print("SwiftExample deinit")
    }
}

func test() {
    swiftTest()
    kotlinTest()
}

func swiftTest() {
    print(SwiftExample())
    print("swiftTestFinished")
}

func kotlinTest() {
    KotlinExample().action(arg: SwiftExample())
    print("kotlinTest finished")
}
```

The resulting output:

```
shared.SwiftExample
SwiftExample deinit
swiftTestFinished
shared.SwiftExample
kotlinTest finished
SwiftExample deinit
```

Objective-C objects lifecycle

The Objective-C objects might live longer than they should, which sometimes might cause performance issues. For example, when a long-running loop creates several temporary objects that cross the Swift/Objective-C interop boundary on each iteration.

In the GC logs, there's a number of stable refs in the root set. If this number keeps growing, it may indicate that the Swift/Objective-C objects are not freed up when they should. In this case, try the autoreleasepool block around loop bodies that do interop calls:

```
// Kotlin
fun growingMemoryUsage() {
    repeat(Int.MAX_VALUE) {
        NSLog("$it\n")
    }
}

fun steadyMemoryUsage() {
    repeat(Int.MAX_VALUE) {
        autoreleasepool {
            NSLog("$it\n")
        }
    }
}
```

Garbage collection of Swift and Kotlin objects' chains

Consider the following example:

```
// Kotlin
interface Storage {
    fun store(arg: Any)
}

class KotlinStorage(var field: Any? = null) : Storage {
    override fun store(arg: Any) {
        field = arg
    }
}
```

```kotlin
class KotlinExample {
    fun action(firstSwiftStorage: Storage, secondSwiftStorage: Storage) {
        // Here, we create the following chain:
        // firstKotlinStorage -> firstSwiftStorage -> secondKotlinStorage -> secondSwiftStorage.
        val firstKotlinStorage = KotlinStorage()
        firstKotlinStorage.store(firstSwiftStorage)
        val secondKotlinStorage = KotlinStorage()
        firstSwiftStorage.store(secondKotlinStorage)
        secondKotlinStorage.store(secondSwiftStorage)
    }
}
```

```swift
// Swift
class SwiftStorage : Storage {

    let name: String

    var field: Any? = nil

    init(_ name: String) {
        self.name = name
    }

    func store(arg: Any) {
        field = arg
    }

    deinit {
        print("deinit SwiftStorage \(name)")
    }
}

func test() {
    KotlinExample().action(
        firstSwiftStorage: SwiftStorage("first"),
        secondSwiftStorage: SwiftStorage("second")
    )
}
```

It takes some time between "deinit SwiftStorage first" and "deinit SwiftStorage second" messages to appear in the log. The reason is that firstKotlinStorage and secondKotlinStorage are collected in different GC cycles. Here's the sequence of events:

1. KotlinExample.action finishes. firstKotlinStorage is considered "dead" because nothing references it, while secondKotlinStorage is not because it is referenced by firstSwiftStorage.
2. First GC cycle starts, and firstKotlinStorage is collected.
3. There are no references to firstSwiftStorage, so it is "dead" as well, and deinit is called.
4. Second GC cycle starts. secondKotlinStorage is collected because firstSwiftStorage is no longer referencing it.
5. secondSwiftStorage is finally reclaimed.

It requires two GC cycles to collect these four objects because deinitialization of Swift and Objective-C objects happens after the GC cycle. The limitation stems from deinit, which can call arbitrary code, including the Kotlin code that cannot be run during the GC pause.

Support for background state and App Extensions

The current memory manager does not track application state by default and does not integrate with App Extensions out of the box.

It means that the memory manager doesn't adjust GC behavior accordingly, which might be harmful in some cases. To change this behavior, add the following Experimental binary option to your gradle.properties:

```
kotlin.native.binary.appStateTracking=enabled
```

It turns off a timer-based invocation of the garbage collector when the application is in the background, so GC is called only when memory consumption becomes too high.

Migrate to the new memory manager

This guide compares the new Kotlin/Native memory manager with the legacy one and describes how to migrate your projects.

The most noticeable change in the new memory manager is lifting restrictions on object sharing. You don't need to freeze objects to share them between threads, specifically:

- Top-level properties can be accessed and modified by any thread without using @SharedImmutable.
- Objects passing through interop can be accessed and modified by any thread without freezing them.
- Worker.executeAfter no longer requires operations to be frozen.
- Worker.execute no longer requires producers to return an isolated object subgraph.
- Reference cycles containing AtomicReference and FreezableAtomicReference do not cause memory leaks.

Apart from easy object sharing, the new memory manager also brings other major changes:

- Global properties are initialized lazily when the file they are defined in is accessed first. Previously global properties were initialized at the program startup. As a workaround, you can mark properties that must be initialized at the program start with the @EagerInitialization annotation. Before using, check its documentation.
- by lazy {} properties support thread-safety modes and do not handle unbounded recursion.
- Exceptions that escape operation in Worker.executeAfter are processed like in other runtime parts, by trying to execute a user-defined unhandled exception hook or terminating the program if the hook was not found or failed with an exception itself.
- Freezing is deprecated, disabled by default, and will be removed in future releases. Do not use freezing if you don't need your code to work with the legacy memory manager.

Follow these guidelines to migrate your projects from the legacy memory manager:

Update Kotlin

The new Kotlin/Native memory manager has been enabled by default since Kotlin 1.7.20. Check the Kotlin version and update to the latest one if necessary.

Update dependencies

kotlinx.coroutines

Update to version 1.6.0 or later. Do not use versions with the native-mt suffix.

There are also some specifics with the new memory manager you should keep in mind:

- Every common primitive (channels, flows, coroutines) works through the Worker boundaries, since freezing is not required.
- Dispatchers.Default is backed by a pool of Workers on Linux and Windows and by a global queue on Apple targets.
- Use newSingleThreadContext to create a coroutine dispatcher that is backed by a Worker.
- Use newFixedThreadPoolContext to create a coroutine dispatcher backed by a pool of N Workers.
- Dispatchers.Main is backed by the main queue on Darwin and by a standalone Worker on other platforms.

Ktor

Update to version 2.0 or later.

Other dependencies

The majority of libraries should work without any changes, however, there might be exceptions.

Make sure that you update dependencies to the latest versions, and there is no difference between library versions for the legacy and the new memory manager.

Update your code

To support the new memory manager, remove usages of the affected API:

Old API	What to do
@SharedImmutable	You can remove all usages, though there are no warnings for using this API in the new memory manager.
The FreezableAtomicReference class	Use AtomicReference instead.
The FreezingException class	Remove all usages.
The InvalidMutabilityException class	Remove all usages.
The IncorrectDereferenceException class	Remove all usages.
The freeze() function	Remove all usages.
The isFrozen property	You can remove all usages. Since freezing is deprecated, the property always returns false.
The ensureNeverFrozen() function	Remove all usages.
The atomicLazy() function	Use lazy() instead.
The MutableData class	Use any regular collection instead.
The WorkerBoundReference<out T : Any> class	Use T directly.
The DetachedObjectGraph<T> class	Use T directly. To pass the value through the C interop, use the StableRef class.

Support both new and legacy memory managers

If you're a library author and need to maintain support for the legacy memory manager or want to have a fallback in case of issues with the new memory manager, you can temporarily support code for both new and legacy memory managers.

To ignore deprecation warnings, do one of the following:

- Annotate usages of the deprecated API with @OptIn(FreezingIsDeprecated::class).
- Apply languageSettings.optIn("kotlin.native.FreezingIsDeprecated") to all the Kotlin source sets in Gradle.
- Pass the compiler flag -opt-in=kotlin.native.FreezingIsDeprecated.

See Opt-in requirements for more details.

What's next

- Learn about the new memory manager
- Configure integration with iOS

Immutability and concurrency in Kotlin/Native

This page describes the features of the legacy memory manager. Check out Kotlin/Native memory management to learn about the new memory manager, which has been enabled by default since Kotlin 1.7.20.

Kotlin/Native implements strict mutability checks, ensuring the important invariant that the object is either immutable or accessible from the single thread at that moment in time (mutable XOR global).

Immutability is a runtime property in Kotlin/Native, and can be applied to an arbitrary object subgraph using the kotlin.native.concurrent.freeze function. It makes all the objects reachable from the given one immutable. Such a transition is a one-way operation. For example, objects cannot be unfrozen later. Some naturally immutable objects such as kotlin.String, kotlin.Int, and other primitive types, along with AtomicInt and AtomicReference, are frozen by default. If a mutating operation is applied to a frozen object, an InvalidMutabilityException is thrown.

To achieve mutable XOR global invariant, all globally visible states (currently, object singletons and enums) are automatically frozen. If object freezing is not desired, a kotlin.native.ThreadLocal annotation can be used, which will make the object state thread-local, and so, mutable (but the changed state is not visible to other threads).

Top-level/global variables of non-primitive types are by default accessible in the main thread (i.e., the thread which initialized Kotlin/Native runtime first) only. Access from another thread will lead to an IncorrectDereferenceException being thrown. To make such variables accessible in other threads, you can use either the @ThreadLocal annotation and mark the value thread-local or @SharedImmutable, which will make the value frozen and accessible from other threads. See Global variables and singletons.

Class AtomicReference can be used to publish the changed frozen state to other threads and build patterns like shared caches. See Atomic primitives and references.

Concurrency in Kotlin/Native

Kotlin/Native runtime doesn't encourage a classical thread-oriented concurrency model with mutually exclusive code blocks and conditional variables, as this model is known to be error-prone and unreliable. Instead, we suggest a collection of alternative approaches, allowing you to use hardware concurrency and implement blocking IO. Those approaches are as follows, and they will be elaborated on in further sections:

Workers

Instead of threads, Kotlin/Native runtime offers the concept of workers: concurrently executed control flow streams with an associated request queue. Workers are very similar to the actors in the Actor Model. A worker can exchange Kotlin objects with another worker so that at any moment, each mutable object is owned by a single worker, but ownership can be transferred. See section Object transfer and freezing.

Once a worker is started with the Worker.start function call, it can be addressed with its own unique integer worker id. Other workers, or non-worker concurrency primitives, such as OS threads, can send a message to the worker with the execute call.

```
val future = execute(TransferMode.SAFE, { SomeDataForWorker() }) {
   // data returned by the second function argument comes to the
   // worker routine as 'input' parameter.
   input ->
   // Here we create an instance to be returned when someone consumes result future.
   WorkerResult(input.stringParam + " result")
}

future.consume {
  // Here we see result returned from routine above. Note that future object or
  // id could be transferred to another worker, so we don't have to consume future
  // in same execution context it was obtained.
  result -> println("result is $result")
}
```

The call to execute uses a function passed as its second parameter to produce an object subgraph (for example, a set of mutually referring objects) which is then passed as a whole to that worker. It is then no longer available to the thread that initiated the request. This property is checked if the first parameter is TransferMode.SAFE by graph traversal and is just assumed to be true if it is TransferMode.UNSAFE. The last parameter to execute is a special Kotlin lambda, which is not allowed to capture any state and is actually invoked in the target worker's context. Once processed, the result is transferred to whatever consumes it in the future, and it is attached to the object graph of that worker/thread.

If an object is transferred in UNSAFE mode and is still accessible from multiple concurrent executors, the program will likely crash unexpectedly, so consider that last resort in optimizing, not a general-purpose mechanism.

Object transfer and freezing

An important invariant that Kotlin/Native runtime maintains is that the object is either owned by a single thread/worker, or it is immutable (shared XOR mutable). This ensures that the same data has a single mutator, and so there is no need for locking to exist. To achieve such an invariant, we use the concept of not externally referred object subgraphs. This is a subgraph without external references from outside of the subgraph, which could be checked algorithmically with O(N) complexity (in ARC systems), where N is the number of elements in such a subgraph. Such subgraphs are usually produced as a result of a lambda expression, for example, some builder, and may not contain objects referred to externally.

Freezing is a runtime operation making a given object subgraph immutable by modifying the object header so that future mutation attempts throw an InvalidMutabilityException. It is deep, so if an object has a pointer to other objects, the transitive closure of such objects will be frozen. Freezing is a one-way transformation; frozen objects cannot be unfrozen. Frozen objects have a nice property that, due to their immutability, they can be freely shared between multiple workers/threads without breaking the "mutable XOR shared" invariant.

If an object is frozen, it can be checked with an extension property isFrozen, and if it is, object sharing is allowed. Currently, Kotlin/Native runtime only freezes the enum objects after creation, although additional auto-freezing of certain provably immutable objects could be implemented in the future.

Object subgraph detachment

An object subgraph without external references can be disconnected using DetachedObjectGraph<T> to a COpaquePointer value, which could be stored in void* data, so the disconnected object subgraphs can be stored in a C data structure, and later attached back with DetachedObjectGraph<T>.attach() in an arbitrary thread or a worker. Together with raw memory sharing, it allows side-channel object transfer between concurrent threads if the worker mechanisms are insufficient for a particular task. Note that object detachment may require an explicit leaving function holding object references and then performing cyclic garbage collection. For example, code like:

```
val graph = DetachedObjectGraph {
    val map = mutableMapOf<String, String>()
    for (entry in map.entries) {
        // ...
    }
    map
}
```

will not work as expected and will throw runtime exception, as there are uncollected cycles in the detached graph, while:

```
val graph = DetachedObjectGraph {
    {
     val map = mutableMapOf<String, String>()
     for (entry in map.entries) {
         // ...
     }
     map
    }().also {
      kotlin.native.internal.GC.collect()
    }
 }
```

will work properly, as holding references will be released, and then cyclic garbage affecting the reference counter is collected.

Raw shared memory

Considering the strong ties between Kotlin/Native and C via interoperability, in conjunction with the other mechanisms mentioned above, it is possible to build popular data structures, like concurrent hashmap or shared cache, with Kotlin/Native. It is possible to rely upon shared C data and store references to detached object subgraphs in it. Consider the following .def file:

```
package = global

---
typedef struct {
  int version;
  void* kotlinObject;
} SharedData;

SharedData sharedData;
```

After running the cinterop tool, it can share Kotlin data in a versionized global structure, and interact with it from Kotlin transparently via autogenerated Kotlin like this:

```
class SharedData(rawPtr: NativePtr) : CStructVar(rawPtr) {
```

```
    var version: Int
    var kotlinObject: COpaquePointer?
}
```

So in combination with the top-level variable declared above, it can allow looking at the same memory from different threads and building traditional concurrent structures with platform-specific synchronization primitives.

Global variables and singletons

Frequently, global variables are a source of unintended concurrency issues, so Kotlin/Native implements the following mechanisms to prevent the unintended sharing of state via global objects:

- global variables, unless specially marked, can be only accessed from the main thread (that is, the thread Kotlin/Native runtime was first initialized), if other thread access such a global, IncorrectDereferenceException is thrown
- for global variables marked with the @kotlin.native.ThreadLocal annotation, each thread keeps a thread-local copy, so changes are not visible between threads
- for global variables marked with the @kotlin.native.SharedImmutable annotation value is shared, but frozen before publishing, so each thread sees the same value
- singleton objects unless marked with @kotlin.native.ThreadLocal are frozen and shared, lazy values allowed, unless cyclic frozen structures were attempted to be created
- enums are always frozen

These mechanisms combined allow natural race-free programming with code reuse across platforms in Multiplatform projects.

Atomic primitives and references

Kotlin/Native standard library provides primitives for safe working with concurrently mutable data, namely AtomicInt, AtomicLong, AtomicNativePtr, AtomicReference and FreezableAtomicReference in the package kotlin.native.concurrent. Atomic primitives allow concurrency-safe update operations, such as increment, decrement, and compare-and-swap, along with value setters and getters. Atomic primitives are always considered frozen by the runtime, and while their fields can be updated with the regular field.value += 1, it is not concurrency safe. The value must be changed using dedicated operations so it is possible to perform concurrent-safe global counters and similar data structures.

Some algorithms require shared mutable references across multiple workers. For example, the global mutable configuration could be implemented as an immutable instance of properties list atomically replaced with the new version on configuration update as the whole in a single transaction. This way, no inconsistent configuration could be seen, and at the same time, the configuration could be updated as needed. To achieve such functionality, Kotlin/Native runtime provides two related classes: kotlin.native.concurrent.AtomicReference and kotlin.native.concurrent.FreezableAtomicReference. Atomic reference holds a reference to a frozen or immutable object, and its value could be updated by set or compare-and-swap operation. Thus, a dedicated set of objects could be used to create mutable shared object graphs (of immutable objects). Cycles in the shared memory could be created using atomic references. Kotlin/Native runtime doesn't support garbage collecting cyclic data when the reference cycle goes through AtomicReference or frozen FreezableAtomicReference. So to avoid memory leaks, atomic references that are potentially parts of shared cyclic data should be zeroed out once no longer needed.

If atomic reference value is attempted to be set to a non-frozen value, a runtime exception is thrown.

Freezable atomic reference is similar to the regular atomic reference until frozen behaves like a regular box for a reference. After freezing, it behaves like an atomic reference and can only hold a reference to a frozen object.

Concurrency overview

> This page describes the features of the legacy memory manager. Check out Kotlin/Native memory management to learn about the new memory manager, which has been enabled by default since Kotlin 1.7.20.

When you extend your development experience from Android to Kotlin Multiplatform for mobile, you will encounter a different state and concurrency model for iOS. This is a Kotlin/Native model that compiles Kotlin code to native binaries that can run without a virtual machine, for example on iOS.

Having mutable memory available to multiple threads at the same time, if unrestricted, is known to be risky and prone to error. Languages like Java, C++, and Swift/Objective-C let multiple threads access the same state in an unrestricted way. Concurrency issues are unlike other programming issues in that they are often very difficult to reproduce. You may not see them locally while developing, and they may happen sporadically. And sometimes you can only see them in production under load.

In short, just because your tests pass, you can't necessarily be sure that your code is OK.

Not all languages are designed this way. JavaScript simply does not allow you to access the same state concurrently. At the other end of the spectrum is Rust, with its language-level management of concurrency and states, which makes it very popular.

Rules for state sharing

Kotlin/Native introduces rules for sharing states between threads. These rules exist to prevent unsafe shared access to mutable states. If you come from a JVM background and write concurrent code, you may need to change the way you architect your data, but doing so will allow you to achieve the same results without risky side effects.

It is also important to point out that there are ways to work around these rules. The intent is to make working around these rules something that you rarely have to do, if ever.

There are just two simple rules regarding state and concurrency.

Rule 1: Mutable state == 1 thread

If your state is mutable, only one thread can see it at a time. Any regular class state that you would normally use in Kotlin is considered by the Kotlin/Native runtime as mutable. If you aren't using concurrency, Kotlin/Native behaves the same as any other Kotlin code, with the exception of global state.

```
data class SomeData(var count:Int)

fun simpleState(){
    val sd = SomeData(42)
    sd.count++
    println("My count is ${sd.count}") // It will be 43
}
```

If there's only one thread, you won't have concurrency issues. Technically this is referred to as thread confinement, which means that you cannot change the UI from a background thread. Kotlin/Native's state rules formalize that concept for all threads.

Rule 2: Immutable state == many threads

If a state can't be changed, multiple threads can safely access it. In Kotlin/Native, immutable doesn't mean everything is a val. It means frozen state.

Immutable and frozen state

The example below is immutable by definition – it has 2 val elements, and both are of final immutable types.

```
data class SomeData(val s:String, val i:Int)
```

This next example may be immutable or mutable. It is not clear what SomeInterface will do internally at compile time. In Kotlin, it is not possible to determine deep immutability statically at compile time.

```
data class SomeData(val s:String, val i:SomeInterface)
```

Kotlin/Native needs to verify that some part of a state really is immutable at runtime. The runtime could simply go through the whole state and verify that each part is deeply immutable, but that would be inflexible. And if you needed to do that every time the runtime wanted to check mutability, there would be significant consequences for performance.

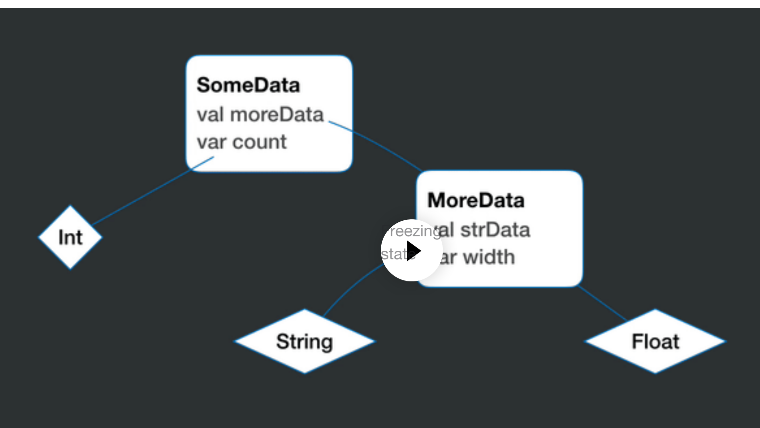
Kotlin/Native defines a new runtime state called frozen. Any instance of an object may be frozen. If an object is frozen:

1. You cannot change any part of its state. Attempting to do so will result in a runtime exception: InvalidMutabilityException. A frozen object instance is 100%, runtime-verified, immutable.

2. Everything it references is also frozen. All other objects it has a reference to are guaranteed to be frozen. This means that, when the runtime needs to determine whether an object can be shared with another thread, it only needs to check whether that object is frozen. If it is, the whole graph is also frozen and is safe to be shared.

The Native runtime adds an extension function freeze() to all classes. Calling freeze() will freeze an object, and everything referenced by the object, recursively.

```
data class MoreData(val strData: String, var width: Float)
data class SomeData(val moreData: MoreData, var count: Int)
//...
val sd = SomeData(MoreData("abc", 10.0), 0)
sd.freeze()
```



Watch animation online.

- freeze() is a one-way operation. You can't unfreeze something.
- freeze() is not available in shared Kotlin code, but several libraries provide expect and actual declarations for using it in shared code. However, if you're using a concurrency library, like kotlinx.coroutines, it will likely freeze data that crosses thread boundaries automatically.

freeze is not unique to Kotlin. You can also find it in Ruby and JavaScript.

Global state

Kotlin allows you to define a state as globally available. If left simply mutable, the global state would violate Rule 1.
To conform to Kotlin/Native's state rules, the global state has some special conditions. These conditions freeze the state or make it visible only to a single thread.

Global object

Global object instances are frozen by default. This means that all threads can access them, but they are immutable. The following won't work.

```
object SomeState{
    var count = 0
    fun add(){
        count++ //This will throw an exception
    }
}
```

Trying to change count will throw an exception because SomeState is frozen (which means all of its data is frozen).

You can make a global object thread local, which will allow it to be mutable and give each thread a copy of its state. Annotate it with @ThreadLocal.

```
@ThreadLocal
object SomeState{
    var count = 0
    fun add(){
        count++ //
    }
}
```

If different threads read count, they'll get different values, because each thread has its own copy.

These global object rules also apply to companion objects.

```
class SomeState{
    companion object{
```

```
        var count = 0
        fun add(){
            count++ //This will throw an exception
        }
    }
}
```

Global properties

Global properties are a special case. They are only available to the main thread, but they are mutable. Accessing them from other threads will throw an exception.

```
val hello = "Hello" //Only main thread can see this
```

You can annotate them with :

- @SharedImmutable, which will make them globally available but frozen.
- @ThreadLocal, which will give each thread its own mutable copy.

This rule applies to global properties with backing fields. Computed properties and global functions do not have the main thread restriction.

Current and future models

Kotlin/Native's concurrency rules will require some adjustment in architecture design, but with the help of libraries and new best practices, day to day development is basically unaffected. In fact, adhering to Kotlin/Native's rules regarding multiplatform code will result in safer concurrency across the cross-platform mobile application.

In the Kotlin Multiplatform application, you have Android and iOS targets with different state rules. Some teams, generally ones working on larger applications, share code for very specific functionality, and often manage concurrency in the host platform. This will require explicit freezing of states returned from Kotlin, but otherwise, it is straightforward.

A more extensive model, where concurrency is managed in Kotlin and the host communicates on its main thread to shared code, is simpler from a state management perspective. Concurrency libraries, like kotlinx.coroutines, will help automate freezing. You'll also be able to leverage the power of coroutines in your code and increase efficiency by sharing more code.

However, the current Kotlin/Native concurrency model has a number of deficiencies. For example, mobile developers are used to freely sharing their objects between threads, and they have already developed a number of approaches and architectural patterns to avoid data races while doing so. It is possible to write efficient applications that do not block the main thread using Kotlin/Native, but the ability to do so comes with a steep learning curve.

That's why we are working on creating a new memory manager and concurrency model for Kotlin/Native that will help us remove these drawbacks. Learn more about where we are going with this.

This material was prepared by Touchlab for publication by JetBrains.

Concurrent mutability

> This page describes the features of the legacy memory manager. Check out Kotlin/Native memory management to learn about the new memory manager, which has been enabled by default since Kotlin 1.7.20.

When it comes to working with iOS, Kotlin/Native's state and concurrency model has two simple rules.

1. A mutable, non-frozen state is visible to only one thread at a time.
2. An immutable, frozen state can be shared between threads.

The result of following these rules is that you can't change global states, and you can't change the same shared state from multiple threads. In many cases, simply changing your approach to how you design your code will work fine, and you don't need concurrent mutability. States were mutable from multiple threads in JVM code, but they didn't need to be.

However, in many other cases, you may need arbitrary thread access to a state, or you may have service objects that should be available to the entire application. Or maybe you simply don't want to go through the potentially costly exercise of redesigning existing code. Whatever the reason, it will not always be feasible to constrain a mutable state to a single thread.

There are various techniques that help you work around these restrictions, each with their own pros and cons:

- Atomics
- Thread-isolated states
- Low-level capabilities

Atomics

Kotlin/Native provides a set of Atomic classes that can be frozen while still supporting changes to the value they contain. These classes implement a special-case handling of states in the Kotlin/Native runtime. This means that you can change values inside a frozen state.

The Kotlin/Native runtime includes a few different variations of Atomics. You can use them directly or from a library.

Kotlin provides an experimental low-level kotlinx.atomicfu library that is currently used only for internal purposes and is not supported for general usage. You can also use Stately, a utility library for multiplatform compatibility with Kotlin/Native-specific concurrency, developed by Touchlab.

AtomicInt/AtomicLong

The first two are simple numerics: AtomicInt and AtomicLong. They allow you to have a shared Int or Long that can be read and changed from multiple threads.

```
object AtomicDataCounter {
    val count = AtomicInt(3)

    fun addOne() {
        count.increment()
    }
}
```

The example above is a global object, which is frozen by default in Kotlin/Native. In this case, however, you can change the value of count. It's important to note that you can change the value of count from any thread.

AtomicReference

AtomicReference holds an object instance, and you can change that object instance. The object you put in AtomicReference must be frozen, but you can change the value that AtomicReference holds. For example, the following won't work in Kotlin/Native:

```
data class SomeData(val i: Int)

object GlobalData {
    var sd = SomeData(0)

    fun storeNewValue(i: Int) {
        sd = SomeData(i) //Doesn't work
    }
}
```

According to the rules of global state, global object values are frozen in Kotlin/Native, so trying to modify sd will fail. You could implement it instead with AtomicReference:

```
data class SomeData(val i: Int)

object GlobalData {
    val sd = AtomicReference(SomeData(0).freeze())

    fun storeNewValue(i: Int) {
        sd.value = SomeData(i).freeze()
    }
}
```

The AtomicReference itself is frozen, which lets it live inside something that is frozen. The data in the AtomicReference instance is explicitly frozen in the code above. However, in the multiplatform libraries, the data will be frozen automatically. If you use the Kotlin/Native runtime's AtomicReference, you should remember to call freeze() explicitly.

AtomicReference can be very useful when you need to share a state. There are some drawbacks to consider, however.

Accessing and changing values in an AtomicReference is very costly performance-wise relative to a standard mutable state. If performance is a concern, you may want to consider using another approach involving a thread-isolated state.

There is also a potential issue with memory leaks, which will be resolved in the future. In situations where the object kept in the AtomicReference has cyclical references, it may leak memory if you don't explicitly clear it out:

- If you have state that may have cyclic references and needs to be reclaimed, you should use a nullable type in the AtomicReference and set it to null explicitly when you're done with it.
- If you're keeping AtomicReference in a global object that never leaves scope, this won't matter (because the memory never needs to be reclaimed during the life of the process).

```
class Container(a:A) {
    val atom = AtomicReference<A?>(a.freeze())

    /**
     * Call when you're done with Container
     */
    fun clear(){
        atom.value = null
    }
}
```

Finally, there's also a consistency concern. Setting/getting values in AtomicReference is itself atomic, but if your logic requires a longer chain of thread exclusion, you'll need to implement that yourself. For example, if you have a list of values in an AtomicReference and you want to scan them first before adding a new one, you'll need to have some form of concurrency management that AtomicReference alone does not provide.

The following won't protect against duplicate values in the list if called from multiple threads:

```
object MyListCache {
    val atomicList = AtomicReference(listOf<String>().freeze())
    fun addEntry(s:String){
        val l = atomicList.value
        val newList = mutableListOf<String>()
        newList.addAll(l)
        if(!newList.contains(s)){
            newList.add(s)
        }
        atomicList.value = newList.freeze()
    }
}
```

You will need to implement some form of locking or check-and-set logic to ensure proper concurrency.

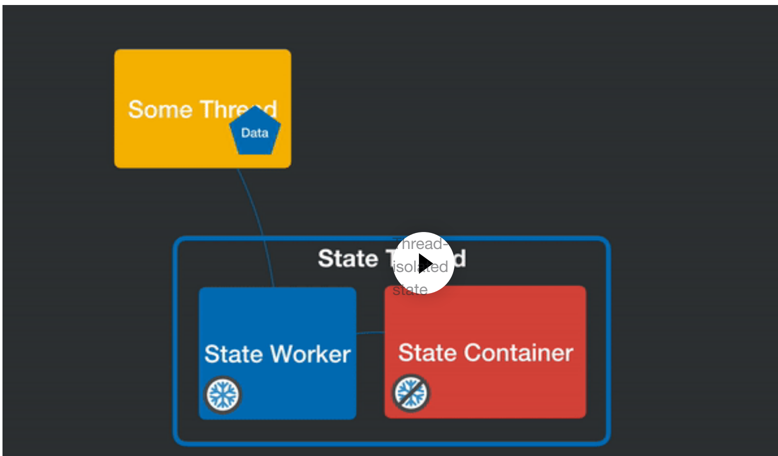
Thread-isolated state

Rule 1 of Kotlin/Native state is that a mutable state is visible to only one thread. Atomics allow mutability from any thread. Isolating a mutable state to a single thread, and allowing other threads to communicate with that state, is an alternative method for achieving concurrent mutability.

To do this, create a work queue that has exclusive access to a thread, and create a mutable state that lives in just that thread. Other threads communicate with the mutable thread by scheduling work on the work queue.

Data that goes in or comes out, if any, needs to be frozen, but the mutable state hidden in the worker thread remains mutable.

Conceptually it looks like the following: one thread pushes a frozen state into the state worker, which stores it in the mutable state container. Another thread later schedules work that takes that state out.



Watch animation online.

Implementing thread-isolated states is somewhat complex, but there are libraries that provide this functionality.

AtomicReference vs. thread-isolated state

For simple values, AtomicReference will likely be an easier option. For cases with significant states, and potentially significant state changes, using a thread-isolated state may be a better choice. The main performance penalty is actually crossing over threads. But in performance tests with collections, for example, a thread-isolated state significantly outperforms a mutable state implemented with AtomicReference.

The thread-isolated state also avoids the consistency issues that AtomicReference has. Because all operations happen in the state thread, and because you're scheduling work, you can perform operations with multiple steps and guarantee consistency without managing thread exclusion. Thread isolation is a design feature of the Kotlin/Native state rules, and isolating mutable states works with those rules.

The thread-isolated state is also more flexible insofar as you can make mutable states concurrent. You can use any type of mutable state, rather than needing to create complex concurrent implementations.

Low-level capabilities

Kotlin/Native has some more advanced ways of sharing concurrent states. To achieve high performance, you may need to avoid the concurrency rules altogether.

> This is a more advanced topic. You should have a deep understanding of how concurrency in Kotlin/Native works under the hood, and you'll need to be very careful when using this approach. Learn more about concurrency.

Kotlin/Native runs on top of C++ and provides interop with C and Objective-C. If you are running on iOS, you can also pass lambda arguments into your shared code from Swift. All of this native code runs outside of the Kotlin/Native state restrictions.

That means that you can implement a concurrent mutable state in a native language and have Kotlin/Native talk to it.

You can use Objective-C interop to access low-level code. You can also use Swift to implement Kotlin interfaces or pass in lambdas that Kotlin code can call from any thread.

One of the benefits of a platform-native approach is performance. On the negative side, you'll need to manage concurrency on your own. Objective-C does not know about frozen, but if you store states from Kotlin in Objective-C structures, and share them between threads, the Kotlin states definitely need to be frozen. Kotlin/Native's runtime will generally warn you about issues, but it's possible to cause concurrency problems in native code that are very, very difficult to track down. It is also very easy to create memory leaks.

Since in the Kotlin Multiplatform application you are also targeting the JVM, you'll need alternate ways to implement anything you use platform native code for. This will obviously take more work and may lead to platform inconsistencies.

This material was prepared by Touchlab for publication by JetBrains.

Concurrency and coroutines

> This page describes the features of the legacy memory manager. Check out Kotlin/Native memory management to learn about the new memory manager, which has been enabled by default since Kotlin 1.7.20.

When working with mobile platforms, you may need to write multithreaded code that runs in parallel. For this, you can use the standard kotlinx.coroutines library or its multithreaded version and alternative solutions.

Review the pros and cons of each solution and choose the one that works best for your situation.

Learn more about concurrency, the current approach, and future improvements.

Coroutines

Coroutines are light-weight threads that allow you to write asynchronous non-blocking code. Kotlin provides the kotlinx.coroutines library with a number of high-level coroutine-enabled primitives.

The current version of kotlinx.coroutines, which can be used for iOS, supports usage only in a single thread. You cannot send work to other threads by changing a dispatcher.

For Kotlin 1.9.0, the recommended coroutines version is 1.7.1.

You can suspend execution and do work on other threads while using a different mechanism for scheduling and managing that work. However, this version of kotlinx.coroutines cannot change threads on its own.

There is also another version of kotlinx.coroutines that provides support for multiple threads.

Get acquainted with the main concepts for using coroutines:

- Asynchronous vs. parallel processing
- Dispatcher for changing threads
- Frozen captured data
- Frozen returned data

Asynchronous vs. parallel processing

Asynchronous and parallel processing are different.

Within a coroutine, the processing sequence may be suspended and resumed later. This allows for asynchronous, non-blocking code, without using callbacks or promises. That is asynchronous processing, but everything related to that coroutine can happen in a single thread.

The following code makes a network call using Ktor. In the main thread, the call is initiated and suspended, while another underlying process performs the actual networking. When completed, the code resumes in the main thread.

```
val client = HttpClient()
//Running in the main thread, start a `get` call
client.get<String>("https://example.com/some/rest/call")
//The get call will suspend and let other work happen in the main thread, and resume when the get call completes
```

That is different from parallel code that needs to be run in another thread. Depending on your purpose and the libraries you use, you may never need to use multiple threads.

Dispatcher for changing threads

Coroutines are executed by a dispatcher that defines which thread the coroutine will be executed on. There are a number of ways in which you can specify the dispatcher, or change the one for the coroutine. For example:

```
suspend fun differentThread() = withContext(Dispatchers.Default){
    println("Different thread")
}
```

withContext takes both a dispatcher as an argument and a code block that will be executed by the thread defined by the dispatcher. Learn more about coroutine context and dispatchers.

To perform work on a different thread, specify a different dispatcher and a code block to execute. In general, switching dispatchers and threads works similar to the JVM, but there are differences related to freezing captured and returned data.

Frozen captured data

To run code on a different thread, you pass a functionBlock, which gets frozen and then runs in another thread.

```
fun <R> runOnDifferentThread(functionBlock: () -> R)
```

You will call that function as follows:

```
runOnDifferentThread {
    //Code run in another thread
}
```

As described in the concurrency overview, a state shared between threads in Kotlin/Native must be frozen. A function argument is a state itself, which will be frozen along with anything it captures.

Coroutine functions that cross threads use the same pattern. To allow function blocks to be executed on another thread, they are frozen.

In the following example, the data class instance dc will be captured by the function block and will be frozen when crossing threads. The println statement will print true.

```
val dc = DataClass("Hello")
withContext(Dispatchers.Default) {
    println("${dc.isFrozen}")
}
```

When running parallel code, be careful with the captured state. Sometimes it's obvious when the state will be captured, but not always. For example:

```
class SomeModel(val id:IdRec){
    suspend fun saveData() = withContext(Dispatchers.Default){
        saveToDb(id)
    }
}
```

The code inside saveData runs on another thread. That will freeze id, but because id is a property of the parent class, it will also freeze the parent class.

Frozen returned data

Data returned from a different thread is also frozen. Even though it's recommended that you return immutable data, you can return a mutable state in a way that doesn't allow a returned value to be changed.

```
val dc = withContext(Dispatchers.Default) {
    DataClass("Hello Again")
}

println("${dc.isFrozen}")
```

It may be a problem if a mutable state is isolated in a single thread and coroutine threading operations are used for communication. If you attempt to return data that retains a reference to the mutable state, it will also freeze the data by association.

Learn more about the thread-isolated state.

Multithreaded coroutines

A special branch of the kotlinx.coroutines library provides support for using multiple threads. It is a separate branch for the reasons listed in the future concurrency model blog post.

However, you can still use the multithreaded version of kotlinx.coroutines in production, taking its specifics into account.

The current version for Kotlin 1.9.0 is 1.7.1-native-mt.

To use the multithreaded version, add a dependency for the commonMain source set in build.gradle(.kts):

Kotlin

```kotlin
val commonMain by getting {
    dependencies {
        implementation ("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1-native-mt")
    }
}
```

Groovy

```groovy
commonMain {
    dependencies {
        implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1-native-mt'
    }
}
```

When using other libraries that also depend on kotlinx.coroutines, such as Ktor, make sure to specify the multithreaded version of kotlinx-coroutines. You can do this with strictly:

Kotlin

```kotlin
implementation ("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1-native-mt") {
    version {
        strictly("1.7.1-native-mt")
    }
}
```

Groovy

```groovy
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1-native-mt' {
    version {
        strictly '1.7.1-native-mt'
    }
}
```

Because the main version of kotlinx.coroutines is a single-threaded one, libraries will almost certainly rely on this version. If you see InvalidMutabilityException related to a coroutine operation, it's very likely that you are using the wrong version.

> Using multithreaded coroutines may result in memory leaks. This can be a problem for complex coroutine scenarios under load. We are working on a solution for this.

See a complete example of using multithreaded coroutines in a Kotlin Multiplatform application.

Alternatives to kotlinx-coroutines

There are a few alternative ways to run parallel code.

CoroutineWorker

CoroutinesWorker is a library published by AutoDesk that implements some features of coroutines across threads using the single-threaded version of kotlinx.coroutines.

For simple suspend functions this is a pretty good option, but it does not support Flow and other structures.

Reaktive

Reaktive is an Rx-like library that implements Reactive extensions for Kotlin Multiplatform. It has some coroutine extensions but is primarily designed around RX and threads.

Custom processor

For simpler background tasks, you can create your own processor with wrappers around platform specifics. See a simple example.

Platform concurrency

In production, you can also rely on the platform to handle concurrency. This could be helpful if the shared Kotlin code will be used for business logic or data operations rather than architecture.

To share a state in iOS across threads, that state needs to be frozen. The concurrency libraries mentioned here will freeze your data automatically. You will rarely need to do so explicitly, if ever.

If you return data to the iOS platform that should be shared across threads, ensure that data is frozen before leaving the iOS boundary.

Kotlin has the concept of frozen only for Kotlin/Native platforms including iOS. To make freeze available in common code, you can create expect and actual implementations for freeze, or use stately-common, which provides this functionality. In Kotlin/Native, freeze will freeze your state, while on the JVM it'll do nothing.

To use stately-common, add a dependency for the commonMain source set in build.gradle(.kts):

Kotlin

```
val commonMain by getting {
    dependencies {
        implementation ("co.touchlab:stately-common:1.0.x")
    }
}
```

Groovy

```
commonMain {
    dependencies {
        implementation 'co.touchlab:stately-common:1.0.x'
    }
}
```

This material was prepared by Touchlab for publication by JetBrains.

Debugging Kotlin/Native

Currently, the Kotlin/Native compiler produces debug info compatible with the DWARF 2 specification, so modern debugger tools can perform the following operations:

- breakpoints
- stepping
- inspection of type information
- variable inspection

> Supporting the DWARF 2 specification means that the debugger tool recognizes Kotlin as C89, because before the DWARF 5 specification, there is no identifier for the Kotlin language type in specification.

Produce binaries with debug info with Kotlin/Native compiler

To produce binaries with the Kotlin/Native compiler, use the -g option on the command line.

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat - > hello.kt
fun main(args: Array<String>) {
  println("Hello world")
  println("I need your clothes, your boots and your motocycle")
}
```

```
0:b-debugger-fixes:minamoto@unit-703(0)# dist/bin/konanc -g hello.kt -o terminator
KtFile: hello.kt
0:b-debugger-fixes:minamoto@unit-703(0)# lldb terminator.kexe
(lldb) target create "terminator.kexe"
Current executable set to 'terminator.kexe' (x86_64).
(lldb) b kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2, address = 0x00000001000012e4
(lldb) r
Process 28473 launched: '/Users/minamoto/ws/.git-trees/debugger-fixes/terminator.kexe' (x86_64)
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000012e4 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at hello.kt:2
   1    fun main(args: Array<String>) {
-> 2      println("Hello world")
   3      println("I need your clothes, your boots and your motocycle")
   4    }
(lldb) n
Hello world
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x00000001000012f0 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at hello.kt:3
   1    fun main(args: Array<String>) {
   2      println("Hello world")
-> 3      println("I need your clothes, your boots and your motocycle")
   4    }
(lldb)
```

Breakpoints

Modern debuggers provide several ways to set a breakpoint, see below for a tool-by-tool breakdown:

lldb

- by name

```
(lldb) b -n kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 4: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2, address = 0x00000001000012e4
```

-n is optional, this flag is applied by default

- by location (filename, line number)

```
(lldb) b -f hello.kt -l 1
Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2, address = 0x00000001000012e4
```

- by address

```
(lldb) b -a 0x00000001000012e4
Breakpoint 2: address = 0x00000001000012e4
```

- by regex, you might find it useful for debugging generated artifacts, like lambda etc. (where used # symbol in name).

```
3: regex = 'main\(', locations = 1
  3.1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2, address =
terminator.kexe[0x00000001000012e4], unresolved, hit count = 0
```

gdb

- by regex

```
(gdb) rbreak main(
Breakpoint 1 at 0x1000109b4
struct ktype:kotlin.Unit &kfun:main(kotlin.Array<kotlin.String>);
```

- by name unusable, because : is a separator for the breakpoint by location

```
(gdb) b kfun:main(kotlin.Array<kotlin.String>)
No source file named kfun.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (kfun:main(kotlin.Array<kotlin.String>)) pending
```

- by location

```
(gdb) b hello.kt:1
Breakpoint 2 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 1.
```

- by address

```
(gdb) b *0x100001704
Note: breakpoint 2 also set at pc 0x100001704.
Breakpoint 3 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 2.
```

Stepping

Stepping functions works mostly the same way as for C/C++ programs.

Variable inspection

Variable inspections for var variables works out of the box for primitive types. For non-primitive types there are custom pretty printers for lldb in konan_lldb.py:

```
λ cat main.kt | nl
     1  fun main(args: Array<String>) {
     2      var x = 1
     3      var y = 2
     4      var p = Point(x, y)
     5      println("p = $p")
     6  }

     7  data class Point(val x: Int, val y: Int)

λ lldb ./program.kexe -o 'b main.kt:5' -o
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b main.kt:5
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 289 at main.kt:5, address = 0x000000000040af11
(lldb) r
Process 4985 stopped
* thread #1, name = 'program.kexe', stop reason = breakpoint 1.1
    frame #0: program.kexe`kfun:main(kotlin.Array<kotlin.String>) at main.kt:5
   2        var x = 1
   3        var y = 2
   4        var p = Point(x, y)
-> 5        println("p = $p")
   6    }
   7
   8    data class Point(val x: Int, val y: Int)

Process 4985 launched: './program.kexe' (x86_64)
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = 0x00000000007643d8
(lldb) command script import dist/tools/konan_lldb.py
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = [x: ..., y: ...]
(lldb) p p
(ObjHeader *) $2 = [x: ..., y: ...]
(lldb) script lldb.frame.FindVariable("p").GetChildMemberWithName("x").Dereference().GetValue()
'1'
(lldb)
```

Getting representation of the object variable (var) could also be done using the built-in runtime function Konan_DebugPrint (this approach also works for gdb, using a module of command syntax):

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat ../debugger-plugin/1.kt | nl -p
     1  fun foo(a:String, b:Int) = a + b
     2  fun one() = 1
     3  fun main(arg:Array<String>) {
     4    var a_variable = foo("(a_variable) one is ", 1)
     5    var b_variable = foo("(b_variable) two is ", 2)
     6    var c_variable = foo("(c_variable) two is ", 3)
     7    var d_variable = foo("(d_variable) two is ", 4)
     8    println(a_variable)
     9    println(b_variable)
    10    println(c_variable)
    11    println(d_variable)
    12  }
0:b-debugger-fixes:minamoto@unit-703(0)# lldb ./program.kexe -o 'b -f 1.kt -l 9' -o r
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b -f 1.kt -l 9
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 463 at 1.kt:9, address = 0x0000000100000dbf
(lldb) r
(a_variable) one is 1
Process 80496 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100000dbf program.kexe`kfun:main(kotlin.Array<kotlin.String>) at 1.kt:9
   6      var c_variable = foo("(c_variable) two is ", 3)
   7      var d_variable = foo("(d_variable) two is ", 4)
   8      println(a_variable)
-> 9      println(b_variable)
   10     println(c_variable)
   11     println(d_variable)
   12   }

Process 80496 launched: './program.kexe' (x86_64)
(lldb) expression -- (int32_t)Konan_DebugPrint(a_variable)
(a_variable) one is 1(int32_t) $0 = 0
(lldb)
```

Known issues

- performance of Python bindings.

Symbolicating iOS crash reports

Debugging an iOS application crash sometimes involves analyzing crash reports. More info about crash reports can be found in the Apple documentation.

Crash reports generally require symbolication to become properly readable: symbolication turns machine code addresses into human-readable source locations. The document below describes some specific details of symbolicating crash reports from iOS applications using Kotlin.

Producing .dSYM for release Kotlin binaries

To symbolicate addresses in Kotlin code (e.g. for stack trace elements corresponding to Kotlin code) .dSYM bundle for Kotlin code is required.

By default, Kotlin/Native compiler produces .dSYM for release (i.e. optimized) binaries on Darwin platforms. This can be disabled with -Xadd-light-debug=disable compiler flag. At the same time, this option is disabled by default for other platforms. To enable it, use the -Xadd-light-debug=enable compiler option.

Kotlin

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

Groovy

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

In projects created from IntelliJ IDEA or AppCode templates these .dSYM bundles are then discovered by Xcode automatically.

Make frameworks static when using rebuild from bitcode

Rebuilding Kotlin-produced framework from bitcode invalidates the original .dSYM. If it is performed locally, make sure the updated .dSYM is used when symbolicating crash reports.

If rebuilding is performed on App Store side, then .dSYM of rebuilt dynamic framework seems discarded and not downloadable from App Store Connect. In this case, it may be required to make the framework static.

Kotlin

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.withType<org.jetbrains.kotlin.gradle.plugin.mpp.Framework> {
            isStatic = true
        }
    }
}
```

Groovy

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        binaries.withType(org.jetbrains.kotlin.gradle.plugin.mpp.Framework) {
            isStatic = true
        }
    }
}
```

Decode inlined stack frames

Xcode doesn't seem to properly decode stack trace elements of inlined function calls (these aren't only Kotlin inline functions but also functions that are inlined when optimizing machine code). So some stack trace elements may be missing. If this is the case, consider using lldb to process crash report that is already symbolicated by Xcode, for example:

```
$ lldb -b -o "script import lldb.macosx" -o "crashlog file.crash"
```

This command should output crash report that is additionally processed and includes inlined stack trace elements.

More details can be found in LLDB documentation.

Kotlin/Native target support

The Kotlin/Native compiler supports a great number of different targets, though it is hard to provide the same level of support for all of them. This document describes which targets Kotlin/Native supports and breaks them into several tiers depending on how well the compiler supports them.

> We can adjust the number of tiers, the list of supported targets, and their features as we go.

Mind the following terms used in tier tables:

* Gradle target name is a target preset that is used in the Kotlin Multiplatform Gradle plugin to enable the target.
* Target triple is a target name according to the <architecture>-<vendor>-<system>-<abi> structure that is commonly used by compilers.
* Running tests indicates out of the box support for running tests in Gradle and IDE.

  This is only available on a native host for the specific target. For example, you can run macosX64 and iosX64 tests only on macOS x86-64 host.

Tier 1

* The target is regularly tested on CI to be able to compile and run.
* We're doing our best to provide a source and binary compatibility between compiler releases.

Gradle target name	Target triple	Running tests	Description
linuxX64	x86_64-unknown-linux-gnu		Linux on x86_64 platforms
Apple macOS hosts only			
macosX64	x86_64-apple-macos		Apple macOS on x86_64 platforms
macosArm64	aarch64-apple-macos		Apple macOS on Apple Silicon platforms
iosSimulatorArm64	aarch64-apple-ios-simulator		Apple iOS simulator on Apple Silicon platforms
iosX64	x86_64-apple-ios-simulator		Apple iOS simulator on x86-64 platforms

Tier 2

* The target is regularly tested on CI to be able to compile, but may not be automatically tested to be able to run.
* We're doing our best to provide a source and binary compatibility between compiler releases.

Gradle target name	Target triple	Running tests	Description
linuxArm64	aarch64-unknown-linux-gnu		Linux on ARM64 platforms
Apple macOS hosts only			
watchosSimulatorArm64	aarch64-apple-watchos-simulator		Apple watchOS simulator on Apple Silicon platforms
watchosX64	x86_64-apple-watchos-simulator		Apple watchOS 64-bit simulator on x86_64 platforms
watchosArm32	armv7k-apple-watchos		Apple watchOS on ARM32 platforms

Gradle target name	Target triple	Running tests	Description
watchosArm64	arm64_32-apple-watchos		Apple watchOS on ARM64 platforms with ILP32
tvosSimulatorArm64	aarch64-apple-tvos-simulator		Apple tvOS simulator on Apple Silicon platforms
tvosX64	x86_64-apple-tvos-simulator		Apple tvOS simulator on x86_64 platforms
tvosArm64	aarch64-apple-tvos		Apple tvOS on ARM64 platforms
iosArm64	aarch64-apple-ios		Apple iOS and iPadOS on ARM64 platforms

> We're doing our best to move iosArm64 to Tier 1, as it's a crucial target for Kotlin Multiplatform for mobile. To do that, we need first to create a dedicated testing infrastructure because platform limitations make it difficult to run compiler tests on Apple devices.
>
> Meanwhile, we sometimes run tests manually on iOS devices and rely on testing similar targets, like iosSimulatorArm64, which should be sufficient in most cases.

Tier 3

- The target is not guaranteed to be tested on CI.
- We can't promise a source and binary compatibility between different compiler releases, though such changes for these targets are quite rare.

Gradle target name	Target triple	Running tests	Description
androidNativeArm32	arm-unknown-linux-androideabi		Android NDK on ARM32 platforms
androidNativeArm64	aarch64-unknown-linux-android		Android NDK on ARM64 platforms
androidNativeX86	i686-unknown-linux-android		Android NDK on x86 platforms
androidNativeX64	x86_64-unknown-linux-android		Android NDK on x86_64 platforms
mingwX64	x86_64-pc-windows-gnu		64-bit MinGW on Windows 7 and later
Apple macOS hosts only			
watchosDeviceArm64	aarch64-apple-watchos		Apple watchOS on ARM64 platforms

Deprecated targets

The following targets are deprecated since Kotlin 1.8.20 and will be removed in 1.9.20:

- iosArm32
- watchosX86
- wasm32
- mingwX86
- linuxArm32Hfp
- linuxMips32
- linuxMipsel32

For library authors

We don't recommend library authors to test more targets or provide stricter guarantees than the Kotlin/Native compiler does. You can use the following approach when considering support for native targets:

- Support all the targets from tier 1, 2, and 3.
- Regularly test targets from tier 1 and 2 that support running tests out of the box.

The Kotlin team uses this approach in the official Kotlin libraries, for example, kotlinx.coroutines and kotlinx.serialization.

Tips for improving Kotlin/Native compilation times

The Kotlin/Native compiler is constantly receiving updates that improve its performance. With the latest Kotlin/Native compiler and a properly configured build environment, you can significantly improve the compilation times of your projects with Kotlin/Native targets.

Read on for our tips on how to speed up the Kotlin/Native compilation process.

General recommendations

- Use the most recent version of Kotlin. This way you will always have the latest performance improvements.
- Avoid creating huge classes. They take a long time to compile and load during execution.
- Preserve downloaded and cached components between builds. When compiling projects, Kotlin/Native downloads the required components and caches some results of its work to the $USER_HOME/.konan directory. The compiler uses this directory for subsequent compilations, making them take less time to complete.

  When building in containers (such as Docker) or with continuous integration systems, the compiler may have to create the ~/.konan directory from scratch for each build. To avoid this step, configure your environment to preserve ~/.konan between builds. For example, redefine its location using the KONAN_DATA_DIR environment variable.

Gradle configuration

The first compilation with Gradle usually takes more time than subsequent ones due to the need to download the dependencies, build caches, and perform additional steps. You should build your project at least twice to get an accurate reading of the actual compilation times.

Here are some recommendations for configuring Gradle for better compilation performance:

- Increase the Gradle heap size. Add org.gradle.jvmargs=-Xmx3g to gradle.properties. If you use parallel builds, you might need to choose the right number of workers with the org.gradle.workers.max property or the --max-workers command-line option. The default value is the number of CPU processors.
- Build only the binaries you need. Don't run Gradle tasks that build the whole project, such as build or assemble, unless you really need to. These tasks build the same code more than once, increasing the compilation times. In typical cases such as running tests from IntelliJ IDEA or starting the app from Xcode, the Kotlin tooling avoids executing unnecessary tasks.

  If you have a non-typical case or build configuration, you might need to choose the task yourself.

  - linkDebug*: To run your code during development, you usually need only one binary, so running the corresponding linkDebug* task should be enough. Keep

in mind that compiling a release binary (linkRelease*) takes more time than compiling a debug one.

- packForXcode: Since iOS simulators and devices have different processor architectures, it's a common approach to distribute a Kotlin/Native binary as a universal (fat) framework. During local development, it will be faster to build the .framework for only the platform you're using.

  To build a platform-specific framework, call the packForXcode task generated by the Kotlin Multiplatform project wizard.

  > Remember that in this case, you will need to clean the build using ./gradlew clean after switching between the device and the simulator. See this issue for details.

- Don't disable the Gradle daemon without having a good reason to. Kotlin/Native runs from the Gradle daemon by default. When it's enabled, the same JVM process is used and there is no need to warm it up for each compilation.
- Don't use transitiveExport = true. Using transitive export disables dead code elimination in many cases: the compiler has to process a lot of unused code. It increases the compilation time. Use export explicitly for exporting the required projects and dependencies.
- Use the Gradle build caches:
  - Local build cache: Add org.gradle.caching=true to your gradle.properties or run with --build-cache on the command line.
  - Remote build cache in continuous integration environments. Learn how to configure the remote build cache.
- Enable previously disabled features of Kotlin/Native. There are properties that disable the Gradle daemon and compiler caches – kotlin.native.disableCompilerDaemon=true and kotlin.native.cacheKind=none. If you had issues with these features before and added these lines to your gradle.properties or Gradle arguments, remove them and check whether the build completes successfully. It is possible that these properties were added previously to work around issues that have already been fixed.

Windows OS configuration

- Configure Windows Security. Windows Security may slow down the Kotlin/Native compiler. You can avoid this by adding the .konan directory, which is located in %USERPROFILE% by default, to Windows Security exclusions. Learn how to add exclusions to Windows Security.

License files for the Kotlin/Native binaries

Like many other open-source projects, Kotlin relies on third-party code, meaning that the Kotlin project includes some code not developed by JetBrains or the Kotlin programming language contributors. Sometimes it is derived work, such as code rewritten from C++ to Kotlin.

> You can find licenses for the third-party work used in Kotlin in our GitHub repository:
>
> - Kotlin compiler
> - Kotlin/Native

In particular, the Kotlin/Native compiler produces binaries that can include third-party code, data, or derived work. This means that the Kotlin/Native-compiled binaries are subject to the terms and conditions of the third-party licenses.

In practice, if you distribute a Kotlin/Native-compiled final binary, you should always include necessary license files in your binary distribution. The files should be accessible to users of your distribution in a readable form.

Always include the following license files for the corresponding projects:

Project	Files to be included
Kotlin	• Apache license 2.0 • Apache Harmony copyright notice

Project	Files to be included
Apache Harmony	
GWT	
Guava	
libbacktrace	3-clause BSD license with copyright notice
mimalloc	MIT license Always include, unless you use the system memory allocator (the -Xallocator=std compiler option is set). For more information on allocators, see Kotlin/Native memory management
Unicode character database	Unicode license

Specific targets require additional license files:

Project	Targets	Files to be included
MinGW-w64 headers and runtime libraries	mingw*	• MinGW-w64 runtime license • Winpthreads license
Musl (math implementation)	wasm32	Musl copyright notice

None of these libraries require the distributed Kotlin/Native binaries to be open-sourced.

Kotlin/Native FAQ

How do I run my program?

Define a top-level function fun main(args: Array<String>) or just fun main() if you are not interested in passed arguments, please ensure it's not in a package. Also, compiler switch -entry could be used to make any function taking Array<String> or no arguments and return Unit as an entry point.

What is Kotlin/Native memory management model?

Kotlin/Native uses an automated memory management scheme that is similar to what Java or Swift provide.

Learn about the Kotlin/Native memory manager

How do I create a shared library?

Use the -produce dynamic compiler switch, or binaries.sharedLib() in Gradle.

```kotlin
kotlin {
    iosArm64("mylib") {
        binaries.sharedLib()
    }
}
```

It will produce a platform-specific shared object (.so on Linux, .dylib on macOS, and .dll on Windows targets) and a C language header, allowing the use of all public APIs available in your Kotlin/Native program from C/C++ code.

How do I create a static library or an object file?

Use the -produce static compiler switch, or binaries.staticLib() in Gradle.

```kotlin
kotlin {
    iosArm64("mylib") {
        binaries.staticLib()
    }
}
```

It will produce a platform-specific static object (.a library format) and a C language header, allowing you to use all the public APIs available in your Kotlin/Native program from C/C++ code.

How do I run Kotlin/Native behind a corporate proxy?

As Kotlin/Native needs to download a platform specific toolchain, you need to specify -Dhttp.proxyHost=xxx -Dhttp.proxyPort=xxx as the compiler's or gradlew arguments, or set it via the JAVA_OPTS environment variable.

How do I specify a custom Objective-C prefix/name for my Kotlin framework?

Use the -module-name compiler option or matching Gradle DSL statement.

Kotlin

```kotlin
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += listOf("-module-name", "TheName")
        }
    }
}
```

Groovy

```groovy
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += ["-module-name", "TheName"]
        }
    }
}
```

How do I rename the iOS framework?

The default name is for an iOS framework is <project name>.framework. To set a custom name, use the baseName option. This will also set the module name.

```kotlin
kotlin {
    iosArm64("myapp") {
       binaries {
          framework {
              baseName = "TheName"
          }
```

```
        }
    }
}
```

How do I enable bitcode for my Kotlin framework?

By default gradle plugin adds it on iOS target.

- For debug build it embeds placeholder LLVM IR data as a marker.
- For release build it embeds bitcode as data.

Or commandline arguments: -Xembed-bitcode (for release) and -Xembed-bitcode-marker (debug)

Setting this in a Gradle DSL:

```
kotlin {
    iosArm64("myapp") {
        binaries {
            framework {
                // Use "marker" to embed the bitcode marker (for debug builds).
                // Use "disable" to disable embedding.
                embedBitcode("bitcode") // for release binaries.
            }
        }
    }
}
```

These options have nearly the same effect as clang's -fembed-bitcode/-fembed-bitcode-marker and swiftc's -embed-bitcode/-embed-bitcode-marker.

Why do I see InvalidMutabilityException?

> This issue is relevant for the legacy memory manager only. Check out Kotlin/Native memory management to learn about the new memory manager, which has been enabled by default since Kotlin 1.7.20.

It likely happens, because you are trying to mutate a frozen object. An object can transfer to the frozen state either explicitly, as objects reachable from objects on which the kotlin.native.concurrent.freeze is called, or implicitly (i.e. reachable from enum or global singleton object - see the next question).

How do I make a singleton object mutable?

> This issue is relevant for the legacy memory manager only. Check out Kotlin/Native memory management to learn about the new memory manager, which has been enabled by default since Kotlin 1.7.20.

Currently, singleton objects are immutable (i.e. frozen after creation), and it's generally considered good practise to have the global state immutable. If for some reason you need a mutable state inside such an object, use the @konan.ThreadLocal annotation on the object. Also, the kotlin.native.concurrent.AtomicReference class could be used to store different pointers to frozen objects in a frozen object and automatically update them.

How can I compile my project with unreleased versions of Kotlin/Native?

First, please consider trying preview versions.

In case you need an even more recent development version, you can build Kotlin/Native from source code: clone Kotlin repository and follow these steps.

Get started with Kotlin/Wasm in IntelliJ IDEA

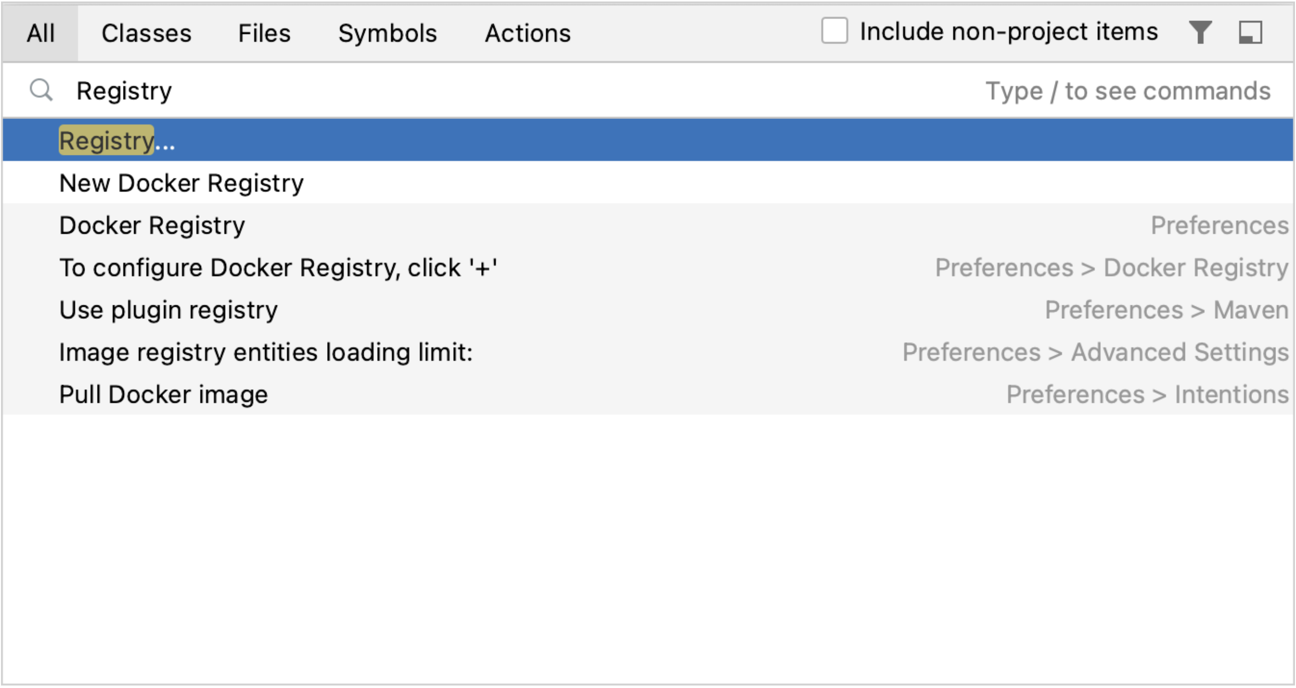
Kotlin/Wasm is an Experimental feature. It may be dropped or changed at any time. It is available only starting with Kotlin 1.8.20.

This tutorial demonstrates how to use IntelliJ IDEA for creating a Kotlin/Wasm application.

To get started, install the latest version of IntelliJ IDEA. The tutorial is applicable to both IntelliJ IDEA Community Edition and the Ultimate Edition.

Enable an experimental Kotlin/Wasm Wizard in IntelliJ IDEA

1. Press double Shift to open a search, enter Registry.



Open registry in IntelliJ IDEA

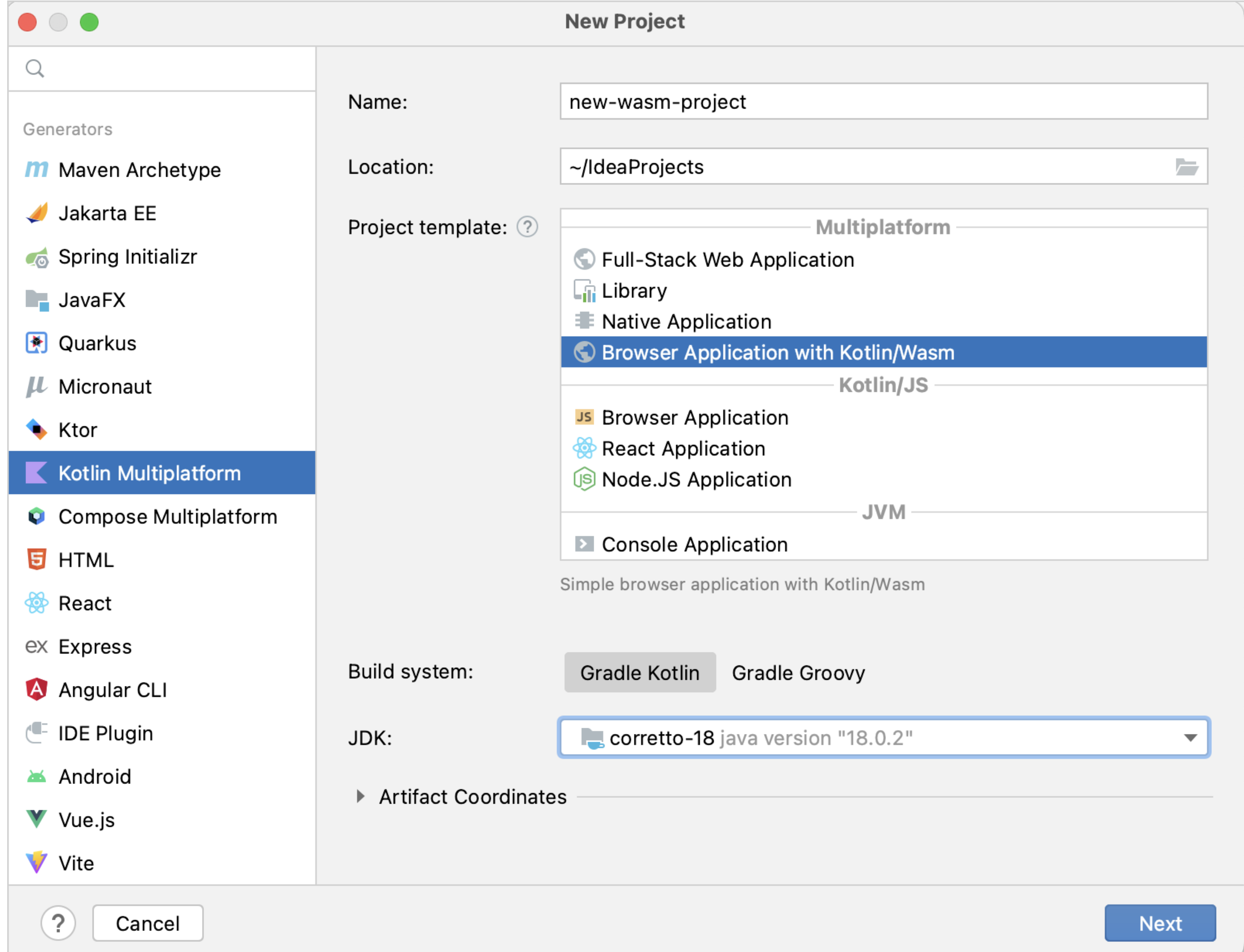
2. Select Registry from the list. Registry window opens.
3. Find the kotlin.wasm.wizard registry key in the list, and enable it.

Enable Kotlin/Wasm Wizard

4. Restart IntelliJ IDEA.

Create a new Kotlin/Wasm project

1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select Kotlin Multiplatform.
3. Enter a project name, select Browser Application with Kotlin/Wasm as the project template, and click Next.



Create a Kotlin/Wasm application

By default, your project will use Gradle with Kotlin DSL as the build system.

4. Accept the default configuration on the next screen and click Finish. Your project will open.

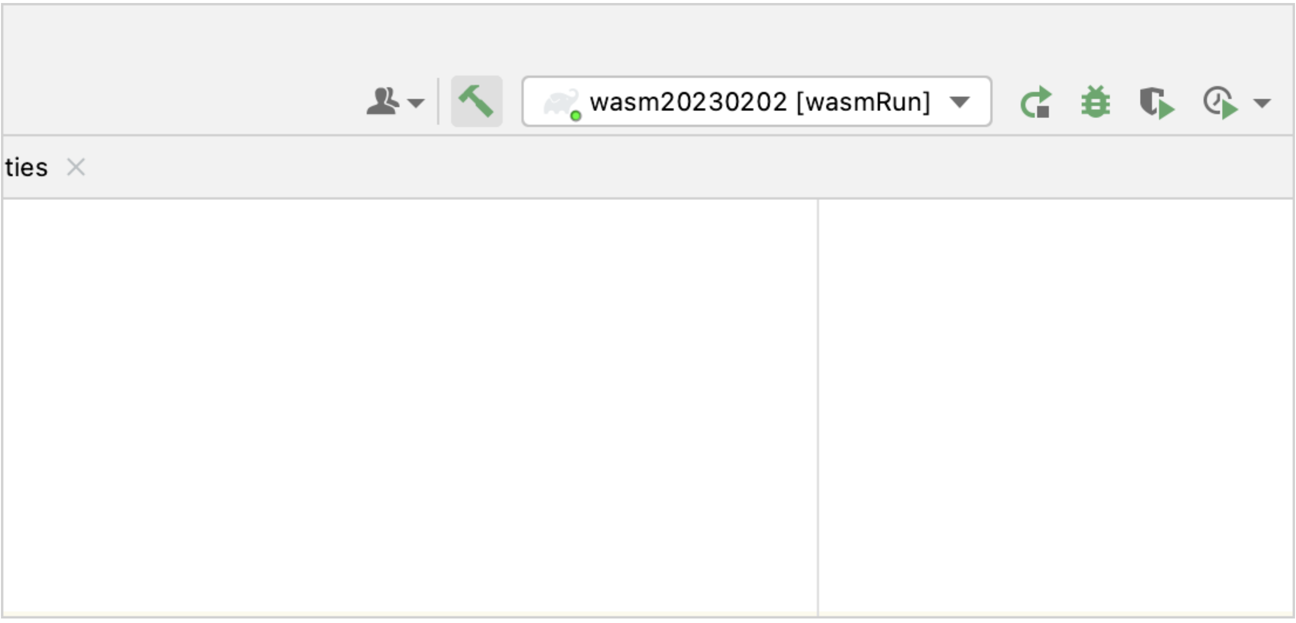
   By default, the wizard creates the necessary Simple.kt file.

5. Open the build.gradle.kts file and ensure that the Kotlin Multiplatform plugin version is set to 1.8.20:

```
plugins {
    kotlin("multiplatform") version "1.8.20"
}
```

Build and run the application

1. Click Build Project next to the run configuration at the top of the screen:



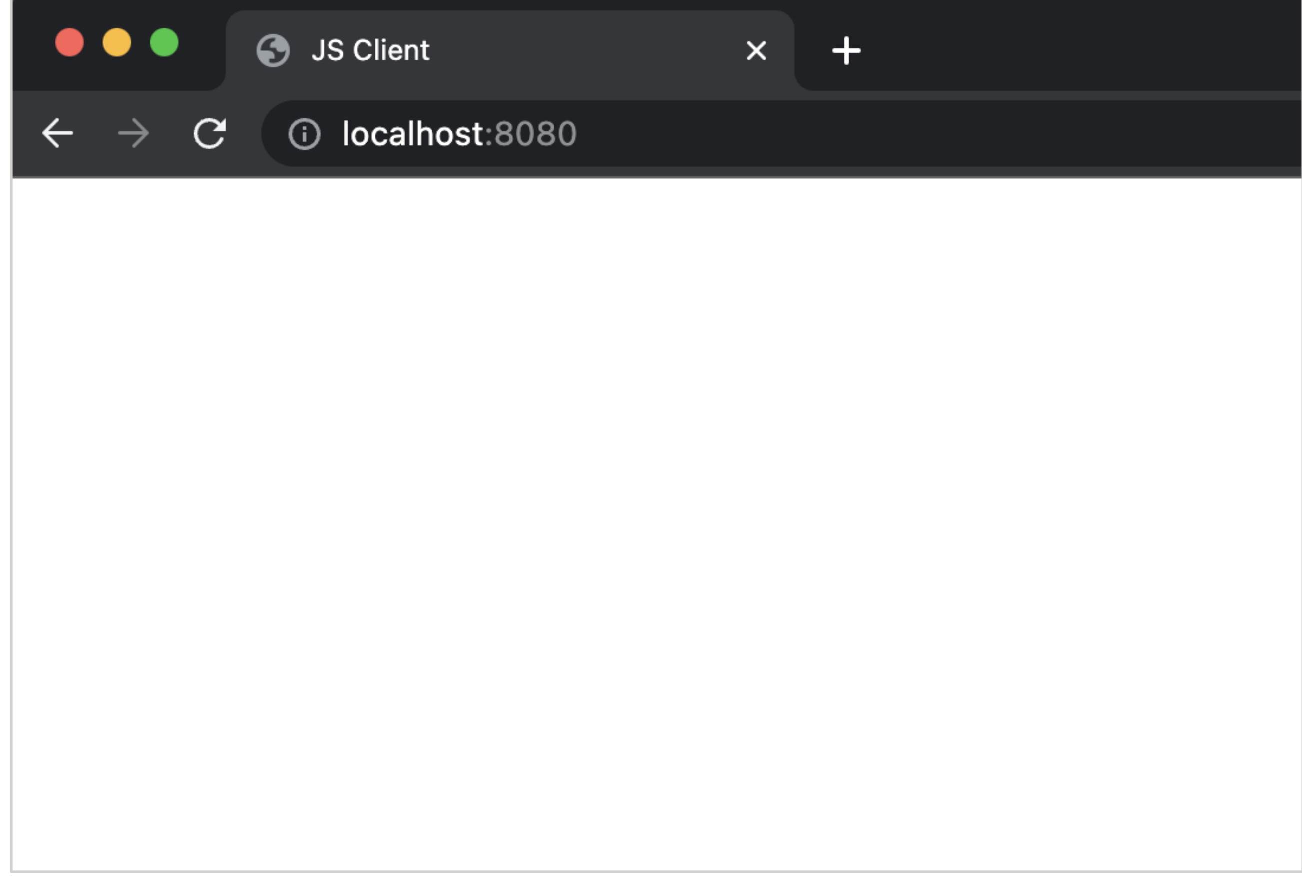
Build the application

2. Run the application by clicking Run next to the run configuration at the top of the screen.

3. Once the application starts, open the following URL in your browser:

```
http://localhost:8080
```

You should see the "JS Client" tab in your browser:



Empty Kotlin/Wasm application in browser

If you open a page source, you'll find the name of the JavaScript bundle:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JS Client</title>
</head>
<body>
<script src="wasm20230202.js"></script>
</body>
</html>
```

Source of Kotlin/Wasm application in browser

Troubleshooting

Despite the fact that most of the browsers support WebAssembly, you need to update the settings in your browser.

To run a Kotlin/Wasm project, you need to update the settings of the target environment:

Chrome

- For version 109:

  Run the application with the --js-flags=--experimental-wasm-gc command line argument.

- For version 110 or later:

  1. Go to chrome://flags/#enable-webassembly-garbage-collection in your browser.
  2. Enable WebAssembly Garbage Collection.
  3. Relaunch your browser.

Firefox

For version 109 or later:

1. Go to about:config in your browser.
2. Enable javascript.options.wasm_function_references and javascript.options.wasm_gc options.
3. Relaunch your browser.

Edge

For version 109 or later:

Run the application with the --js-flags=--experimental-wasm-gc command line argument.

Update your application

1. Open Simple.kt and update the code:

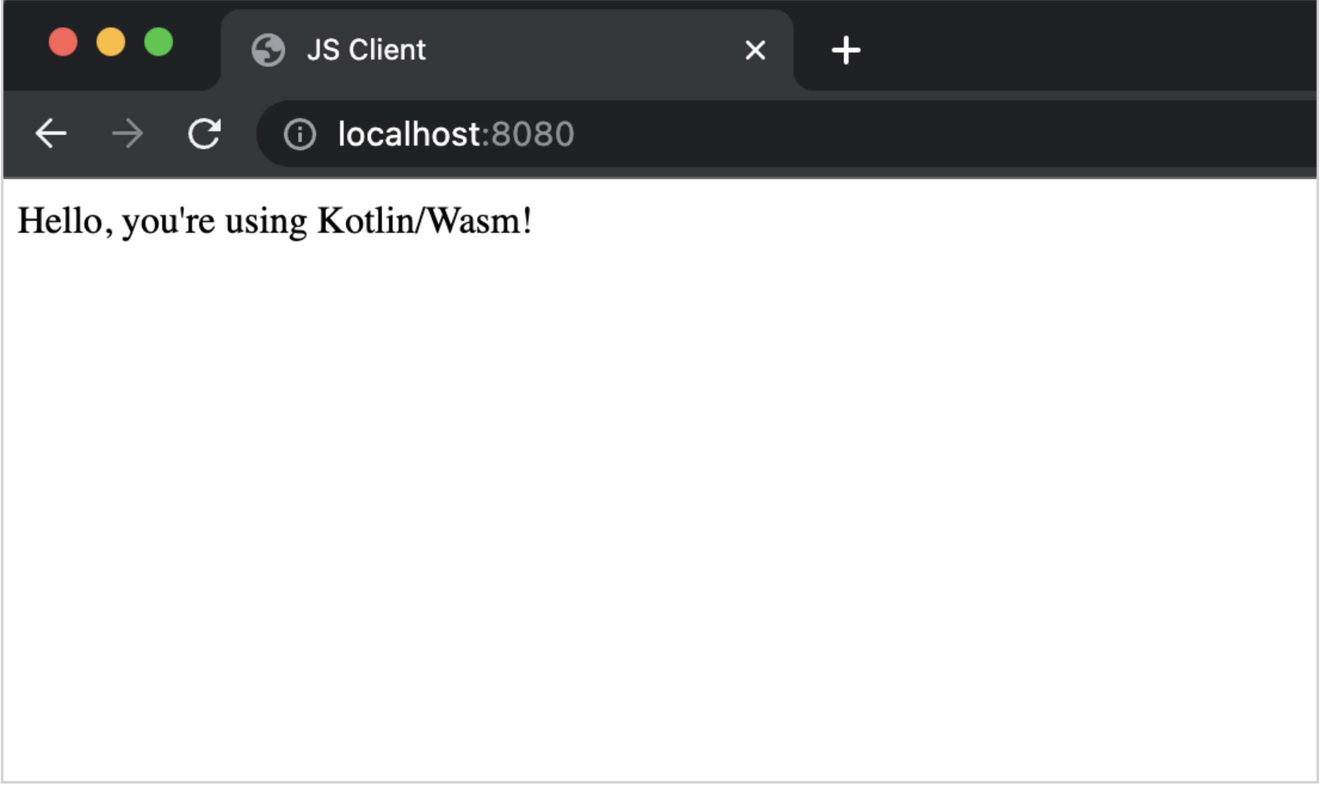
```
import kotlinx.browser.document
import kotlinx.dom.appendText

fun main() {
    println("Hello, ${greet()}")
    document.body!!.appendText("Hello, you're using Kotlin/Wasm!")
}

fun greet() = "world"
```

2. Run the application by clicking Run next to the run configuration at the top of the screen.

3. Once the application starts, open the following URL in your browser:

```
http://localhost:8080
```

You'll see the text "Hello, you're using Kotlin/Wasm!":



Kotlin/Wasm application in browser

What's next?

Explore the Kotlin/Wasm interoperability with JavaScript

Add dependencies on Kotlin libraries to Kotlin/Wasm project

You can use the Kotlin standard library (stdlib) and test library (kotlin.test) in Kotlin/Wasm out of the box. The version of these libraries is the same as the version of the kotlin-multiplatform plugin.

Other official Kotlin (kotlinx) and multiplatform libraries are not fully supported yet. You can try experimental versions of such libraries by adding the Kotlin

experimental repository to your Gradle project.

For Kotlin 1.9.0 and later, use the latest available libraries' versions.

Supported Kotlin libraries for Kotlin/Wasm

You can use one of the following repositories to add Kotlin libraries to your project:

- Maven Central for stdlib and kotlin.test libraries:

```
// build.gradle.kts
repositories {
    mavenCentral()
}
```

- Custom Maven repository for experimental Kotlin/Wasm artifacts:

```
// build.gradle.kts
repositories {
    maven("https://maven.pkg.jetbrains.space/kotlin/p/wasm/experimental")
}
```

- Custom Maven repository for Compose Multiplatform dev artifacts:

```
// build.gradle.kts
repositories {
    maven("https://maven.pkg.jetbrains.space/public/p/compose/dev/")
}
```

Library	Version	Repository
stdlib	1.9.0	Maven Central
kotlin-test	1.9.0	Maven Central
kotlinx-coroutines	1.7.0-RC-wasm0	Custom for experimental Kotlin/Wasm artifacts
Compose Multiplatform	1.4.0-dev-wasm08	Custom for experimental Kotlin/Wasm artifacts
kotlinx-serialization	1.5.1-wasm0	Custom for experimental Kotlin/Wasm artifacts
Ktor	2.3.1-wasm0	Custom for experimental Kotlin/Wasm artifacts
kotlinx-atomicfu	0.20.2-wasm0	Custom for experimental Kotlin/Wasm artifacts
kotlinx-collections-immutable	0.4-wasm0	Custom for experimental Kotlin/Wasm artifacts
kotlinx-datetime	0.4.0-wasm0	Custom for experimental Kotlin/Wasm artifacts

Library	Version	Repository
skiko	0.0.7.61-wasm03	Custom for Compose Multiplatform dev artifacts

Enable libraries in your project

To set a dependency on a library, such as kotlinx.serilization and kotlinx.coroutines, update your build.gradle.kts file:

```
// `build.gradle.kts`

repositories {
    maven("https://maven.pkg.jetbrains.space/kotlin/p/wasm/experimental")
}

kotlin {
    sourceSets {
        val wasmMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-serialization-core-wasm:1.5.1-wasm0")
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-wasm:1.6.4-wasm0")
                implementation("io.ktor:ktor-client-core-wasm:2.3.1-wasm0")
            }
        }
    }
}
```

What's next?

Explore the Kotlin/Wasm interoperability with JavaScript

Interoperability with JavaScript

Kotlin/Wasm allows you to both use JavaScript code from Kotlin and Kotlin code from JavaScript.

The Kotlin/JS compiler already provides the ability to transpile your Kotlin code to JavaScript. The Kotlin/Wasm interoperability with JavaScript is designed in a similar way, taking into account that JavaScript is a dynamically typed language compared to Kotlin. Follow our guide to configure interoperability in your projects.

Remember that Kotlin/Wasm is still Experimental, and some features are not supported yet. We're planning to improve interoperability with JavaScript by implementing some of the missing features or similar functionality.

Use JavaScript code from Kotlin

external modifier

To access JavaScript declarations defined in the global scope, mark them with the external modifier. Consider this JavaScript code sample:

```
// JavaScript

function consoleLogExample() {
    console.log("Hello");
}

let externalInt = 0;

let Counter = {
    value: 0,
    step: 1,
    increment() {
        this.value += this.step;
    }
};
```

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    get area() {
        return this.calcArea();
    }

    calcArea() {
        return this.height * this.width;
    }
}
```

Here's how you can use this JavaScript code in Kotlin:

```
// Kotlin/Wasm

// Use external functions to call JS functions defined in global scope
external fun consoleLogExample(): Unit

// In addition to functions, you can have external top-level properties
external var externalInt: Int

// External objects
external object Counter {
    fun increment(): Unit
    val value: Int
    var step: Int
}

// External class
external class Rectangle(height: Double, width: Double) {
    val height: Double
    val width: Double
    val area: Double
    fun calcArea(): Double
}
```

See the full code in the example project Kotlin/Wasm browser.

> Some "external" Kotlin/JS features are not supported in Kotlin/Wasm:
>
> - Implementing or extending external types
> - External enum classes

@JsFun annotation

To include a small piece of JS code in your Kotlin/Wasm module, use the @JsFun annotation with external top-level functions. The annotation argument should be a string with JS code that evaluates a function with a matching signature:

```
@JsFun("function count(x) { return x + 10; }")
external fun count(x: Int): Int
```

To make it shorter, use arrow syntax:

```
@JsFun("x => x + 10")
external fun count(x: Int): Int
```

The Kotlin compiler doesn't verify these JavaScript snippets and evaluates them as-is. Syntax errors, if any, will be reported when running your JavaScript.

> These function expressions are evaluated only once, before the Wasm module is loaded. Do not rely on side effects as these expressions are not run if the function is not called.

@JsModule

To indicate that an external class, package, function, or property is a JavaScript module, use the @JsModule annotation. Consider this JavaScript code sample:

```
// jsModule.mjs
let maxUsers = 10;

function getActiveUsers() {
    return 10;
};

class User {
    constructor(maxUsers) {
        this.maxUsers = maxUsers;
    }
}

export {maxUsers, getActiveUsers, User};
```

Here's how you can use this JavaScript code in Kotlin:

```
// kotlin
@file:JsModule("./jsModule.mjs")

package example

external val maxUsers: Int
external fun getActiveUsers(): Int
external class User {
    constructor(username: String)
    val username : String
}
```

> Kotlin/Wasm supports ES modules only. That's why you can't use the @JsNonModule annotation.

Use Kotlin code from JavaScript

@JsExport annotation

To make the Kotlin/Wasm declaration available from JavaScript, use the @JsExport annotation with external top-level functions:

```
// Kotlin/Wasm

@JsExport
fun addOne(x: Int): Int = x + 1
```

Now you can use this function from JavaScript in the following way:

```
// JavaScript

import exports from "module.mjs"
exports.addOne(10)
```

Functions marked at @JsExport are visible as properties on a default export of the generated .mjs module. Kotlin types in JavaScript In Kotlin/JS, values are implemented internally as JavaScript primitives and objects. They are passed to and from JavaScript without wrapping or copying.

However, in Kotlin/Wasm, objects have a different representation and are not interchangeable with JavaScript. When you pass a Kotlin object to JavaScript, it's considered as an empty opaque object by default.

The only thing you can do is store it and pass Kotlin objects back to Wasm. However, for primitive types, Kotlin/Wasm can adapt these values so that they can be useful in JavaScript by either copying or wrapping. For efficiency purposes, this is done statically. It's important that these special concrete types are present in function signatures. For example:

```
external fun convertIntAndString(num: Int, text: String)
external fun convertAnyAndChars(num: Any, text: CharSequence)

// ...
```

```
convertIntAndString(10, "Hello") // Converts Int and String to JS Number and String

convertAnyAndChars(10, "Hello") // No conversion
                                // values are passed as opaque references to Wasm objects
```

Kotlin types in JavaScript

Supported types

See how Kotlin types are mapped to JavaScript ones:

Kotlin	JavaScript	Comments
Byte, Short, Int, Char	Number	
Float, Double	Number	
Long	BigInt	
Boolean	Boolean	
String	String	String content is copied. In the future, the stringref proposal could allow the zero-copy string interop.
Unit	Undefined	Only when non-nullable and in functions returning position.
Function type, for example (int, String) → Int	Function reference	Parameters and return values of function types follow the same type of conversion rules.
external interface	Any JS value with given properties	
external class or external object	Corresponding JS class	
Other Kotlin types	Not supported	This includes type Any, arrays, the Throwable class, collections, and so on.
Nullable Type?	Type / null / undefined	
Type parameters <T : U>	Same as the upper bound	In interop declarations, only external types, like JsAny, are supported as upper bounds of type parameters.

Exception handling

The Kotlin/Wasm try-catch expression can't catch the JavaScript exceptions.

If you try to use JavaScript try-catch expression to catch the Kotlin/Wasm exceptions, it'll look like a generic WebAssembly.Exception without directly accessible

messages and data.

Workarounds for Kotlin/JS features non-supported in Kotlin/Wasm

Dynamic type

Kotlin/JS dynamic type used for interoperability with untyped or loosely typed objects is not supported yet. In many cases, you can use external interfaces and the @JsFun annotation instead:

```
// Kotlin/JS

val user: dynamic
val age: Int = 0
user.profile.updateAge(age);

// Kotlin/Wasm

external interface User

@JsFun("(user, age) => user.profile.updateAge(age)")
external fun updateUserAge(user: User, age: Int)

val user: User
val age: Int = 0
updateUserAge(user, age);
```

Inline JavaScript

The js() function used to inline JavaScript code to Kotlin code is not supported yet. Use the @JsFun annotation instead:

```
// Kotlin/JS

fun jsTypeOf(obj: Any): String {
    return js("typeof obj")
}

// Kotlin/Wasm
@JsFun("(obj) => typeof obj")
external fun jsTypeOf(obj: SomeExternalInterfaceType): String
```

Extending external interfaces and classes with non-external classes

Extending JavaScript classes and using external interfaces is not supported yet. Use the @JsFun annotation instead:

```
external interface DataProcessor {
    fun processData(input: String): String
    fun processResult(input: String): String
}

class DataHandler(val handlerData: String) {
    fun processData(input: String): String = input + handlerData
    fun processResult(input: String): String = handlerData + input
}

@JsFun("(processData, processResult) => ({ processData, processResult })")
external fun createDataProcessor(
    processData: (String) -> String,
    processResult: (String) -> String
): DataProcessor

fun convertHandlerToProcessor(handler: DataHandler): DataProcessor =
    createDataProcessor(
        processData = { input -> handler.processData(input) },
        processResult = { input -> handler.processResult(input) }
    )
```

Set up a Kotlin/JS project

Kotlin/JS projects use Gradle as a build system. To let developers easily manage their Kotlin/JS projects, we offer the kotlin.multiplatform Gradle plugin that provides project configuration tools together with helper tasks for automating routines typical for JavaScript development. For example, the plugin downloads the Yarn package manager for managing npm dependencies in background and can build a JavaScript bundle from a Kotlin project using webpack. Dependency management and configuration adjustments can be done to a large part directly from the Gradle build file, with the option to override automatically generated configurations for full control.

You can apply the org.jetbrains.kotlin.multiplatform plugin to a Gradle project manually in the build.gradle(.kts) file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}
```

The Kotlin/JS Gradle plugin lets you manage aspects of your project in the kotlin section of the build script.

```
kotlin {
    //...
}
```

Inside the kotlin section, you can manage the following aspects:

- Target execution environment: browser or Node.js
- Project dependencies: Maven and npm
- Run configuration
- Test configuration
- Bundling and CSS support for browser projects
- Target directory and module name
- Project's package.json file

Execution environments

Kotlin/JS projects can target two different execution environments:

- Browser for client-side scripting in browsers
- Node.js for running JavaScript code outside of a browser, for example, for server-side scripting.

To define the target execution environment for a Kotlin/JS project, add the js section with browser {} or nodejs {} inside.

```
kotlin {
    js {
        browser {
        }
        binaries.executable()
    }
}
```

The instruction binaries.executable() explicitly instructs the Kotlin compiler to emit executable .js files. This is the default behavior when using the current Kotlin/JS compiler, but the instruction is explicitly required if you are working with the Kotlin/JS IR compiler, or have set kotlin.js.generate.executable.default=false in your gradle.properties. In those cases, omitting binaries.executable() will cause the compiler to only generate Kotlin-internal library files, which can be used from other projects, but not run on their own. (This is typically faster than creating executable files, and can be a possible optimization when dealing with non-leaf modules of your project.)

The Kotlin/JS plugin automatically configures its tasks for working with the selected environment. This includes downloading and installing the required environment and dependencies for running and testing the application. This allows developers to build, run, and test simple projects without additional configuration. For projects targeting Node.js, there are also an option to use an existing Node.js installation. Learn how to use pre-installed Node.js.

Dependencies

Like any other Gradle projects, Kotlin/JS projects support traditional Gradle dependency declarations in the dependencies section of the build script.

Kotlin

```
dependencies {
    implementation("org.example.myproject", "1.1.0")
}
```

Groovy

```
dependencies {
    implementation 'org.example.myproject:1.1.0'
}
```

The Kotlin Multiplatform Gradle plugin also supports dependency declarations for particular source sets in the kotlin section of the build script.

Kotlin

```
kotlin {
    sourceSets {
      val jsMain by getting {
            dependencies {
                implementation("org.example.myproject:1.1.0")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        jsMain {
            dependencies {
                implementation 'org.example.myproject:1.1.0'
            }
        }
    }
}
```

Please note that not all libraries available for the Kotlin programming language are available when targeting JavaScript: Only libraries that include artifacts for Kotlin/JS can be used.

If the library you are adding has dependencies on packages from npm, Gradle will automatically resolve these transitive dependencies as well.

Kotlin standard libraries

The dependencies on the standard library is added automatically. The version of the standard library is the same as the version of the kotlin-multiplatform plugin.

The kotlin.test API is available for multiplatform tests. When you create a multiplatform project, the Project Wizard automatically adds test dependencies to all the source sets.

If you didn't use the Project Wizard to create your project, you can add the dependencies manually:

Kotlin

```kotlin
kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // This brings all the platform dependencies automatically
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings all the platform dependencies automatically
            }
        }
    }
}
```

npm dependencies

In the JavaScript world, the most common way to manage dependencies is npm. It offers the biggest public repository of JavaScript modules.

The Kotlin Multiplatform Gradle plugin lets you declare npm dependencies in the Gradle build script, analogous to how you would declare any other dependencies.

To declare an npm dependency, pass its name and version to the npm() function inside a dependency declaration. You can also specify one or multiple version range based on npm's semver syntax.

Kotlin

```kotlin
dependencies {
    implementation(npm("react", "> 14.0.0 <=16.9.0"))
}
```

Groovy

```groovy
dependencies {
    implementation npm('react', '> 14.0.0 <=16.9.0')
}
```

The plugin uses the Yarn package manager to download and install NPM dependencies. It works out of the box without additional configuration, but you can tune it to specific needs. Learn how to configure Yarn in Kotlin Multiplatform Gradle plugin.

Besides regular dependencies, there are three more types of dependencies that can be used from the Gradle DSL. To learn more about when each type of dependency can best be used, have a look at the official documentation linked from npm:

- devDependencies, via devNpm(...),
- optionalDependencies via optionalNpm(...), and
- peerDependencies via peerNpm(...).

Once an npm dependency is installed, you can use its API in your code as described in Calling JS from Kotlin.

run task

The Kotlin/JS plugin provides a jsRun task that lets you run pure Kotlin/JS projects without additional configuration.

For running Kotlin/JS projects in the browser, this task is an alias for the browserDevelopmentRun task (which is also available in Kotlin multiplatform projects). It uses the webpack-dev-server to serve your JavaScript artifacts. If you want to customize the configuration used by webpack-dev-server, for example adjust the port the server runs on, use the webpack configuration file.

For running Kotlin/JS projects targeting Node.js, use the jsRun task that is an alias for the nodeRun task.

To run a project, execute the standard lifecycle jsRun task, or the alias to which it corresponds:

```
./gradlew jsRun
```

To automatically trigger a re-build of your application after making changes to the source files, use the Gradle continuous build feature:

```
./gradlew jsRun --continuous
```

or

```
./gradlew jsRun -t
```

Once the build of your project has succeeded, the webpack-dev-server will automatically refresh the browser page.

test task

The Kotlin Multiplatform Gradle plugin automatically sets up a test infrastructure for projects. For browser projects, it downloads and installs the Karma test runner with other required dependencies; for Node.js projects, the Mocha test framework is used.

The plugin also provides useful testing features, for example:

- Source maps generation
- Test reports generation
- Test run results in the console

For running browser tests, the plugin uses Headless Chrome by default. You can also choose other browser to run tests in, by adding the corresponding entries inside the useKarma section of the build script:

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useIe()
                    useSafari()
                    useFirefox()
                    useChrome()
                    useChromeCanary()
                    useChromeHeadless()
                    usePhantomJS()
                    useOpera()
                }
            }
        }
        binaries.executable()
        // ...
    }
}
```

Alternatively, you can add test targets for browsers in the gradle.properties file:

```
kotlin.js.browser.karma.browsers=firefox,safari
```

This approach allows you to define a list of browsers for all modules, and then add specific browsers in the build scripts of particular modules.

Please note that the Kotlin Multiplatform Gradle plugin does not automatically install these browsers for you, but only uses those that are available in its execution environment. If you are executing Kotlin/JS tests on a continuous integration server, for example, make sure that the browsers you want to test against are installed.

If you want to skip tests, add the line enabled = false to the testTask.

```
kotlin {
    js {
```

```
        browser {
            testTask {
                enabled = false
            }
        }
        binaries.executable()
        // ...
    }
}
```

To run tests, execute the standard lifecycle check task:

```
./gradlew check
```

To specify environment variables used by your Node.js test runners (for example, to pass external information to your tests, or to fine-tune package resolution), use the environment function with a key-value pair inside the testTask block in your build script:

```
kotlin {
    js {
        nodejs {
            testTask {
                environment("key", "value")
            }
        }
    }
}
```

Karma configuration

The Kotlin Multiplatform Gradle plugin automatically generates a Karma configuration file at build time which includes your settings from the kotlin.js.browser.testTask.useKarma block in your build.gradle(.kts). You can find the file at build/js/packages/projectName-test/karma.conf.js. To make adjustments to the configuration used by Karma, place your additional configuration files inside a directory called karma.config.d in the root of your project. All .js configuration files in this directory will be picked up and are automatically merged into the generated karma.conf.js at build time.

All karma configuration abilities are well described in Karma's documentation.

webpack bundling

For browser targets, the Kotlin/JS plugin uses the widely known webpack module bundler.

webpack version

The Kotlin/JS plugin uses webpack 5.

If you have projects created with plugin versions earlier than 1.5.0, you can temporarily switch back to webpack 4 used in these versions by adding the following line to the project's gradle.properties:

```
kotlin.js.webpack.major.version=4
```

webpack task

The most common webpack adjustments can be made directly via the kotlin.js.browser.webpackTask configuration block in the Gradle build file:

- outputFileName - the name of the webpacked output file. It will be generated in <projectDir>/build/dist/<targetName> after an execution of a webpack task. The default value is the project name.
- output.libraryTarget - the module system for the webpacked output. Learn more about available module systems for Kotlin/JS projects. The default value is umd.

```
webpackTask {
    outputFileName = "mycustomfilename.js"
    output.libraryTarget = "commonjs2"
}
```

You can also configure common webpack settings to use in bundling, running, and testing tasks in the commonWebpackConfig block.

webpack configuration file

The Kotlin Multiplatform Gradle plugin automatically generates a standard webpack configuration file at the build time. It is located in build/js/packages/projectName/webpack.config.js.

If you want to make further adjustments to the webpack configuration, place your additional configuration files inside a directory called webpack.config.d in the root of your project. When building your project, all .js configuration files will automatically be merged into the build/js/packages/projectName/webpack.config.js file. To add a new webpack loader, for example, add the following to a .js file inside the webpack.config.d:

> In this case, the configuration object presented in the config global object. You need to modify it in your script.

```
config.module.rules.push({
    test: /\.extension$/,
    loader: 'loader-name'
});
```

All webpack configuration capabilities are well described in its documentation.

Building executables

For building executable JavaScript artifacts through webpack, the Kotlin/JS plugin contains the browserDevelopmentWebpack and browserProductionWebpack Gradle tasks.

- browserDevelopmentWebpack creates development artifacts, which are larger in size, but take little time to create. As such, use the browserDevelopmentWebpack tasks during active development.
- browserProductionWebpack applies dead code elimination to the generated artifacts and minifies the resulting JavaScript file, which takes more time, but generates executables that are smaller in size. As such, use the browserProductionWebpack task when preparing your project for production use.

Execute either of these tasks to obtain the respective artifacts for development or production. The generated files will be available in build/dist unless specified otherwise.

```
./gradlew browserProductionWebpack
```

Note that these tasks will only be available if your target is configured to generate executable files (via binaries.executable()).

CSS

The Kotlin Multiplatform Gradle plugin also provides support for webpack's CSS and style loaders. While all options can be changed by directly modifying the webpack configuration files that are used to build your project, the most commonly used settings are available directly from the build.gradle(.kts) file.

To turn on CSS support in your project, set the cssSupport.enabled option in the Gradle build file in the commonWebpackConfig block. This configuration is also enabled by default when creating a new project using the wizard.

Kotlin

```
browser {
    commonWebpackConfig {
        cssSupport {
            enabled.set(true)
        }
    }
}
```

Groovy

```
browser {
    commonWebpackConfig {
        cssSupport {
            it.enabled.set(true)
        }
    }
}
```

Alternatively, you can add CSS support independently for webpackTask, runTask, and testTask.

Kotlin

```
browser {
    webpackTask {
        cssSupport {
            enabled.set(true)
        }
    }
    runTask {
        cssSupport {
            enabled.set(true)
        }
    }
    testTask {
        useKarma {
            // ...
            webpackConfig.cssSupport {
                enabled.set(true)
            }
        }
    }
}
```

Groovy

```
browser {
    webpackTask {
        cssSupport {
            it.enabled.set(true)
        }
    }
    runTask {
        cssSupport {
            it.enabled.set(true)
        }
    }
    testTask {
        useKarma {
            // ...
            webpackConfig.cssSupport {
                it.enabled.set(true)
            }
        }
    }
}
```

Activating CSS support in your project helps prevent common errors that occur when trying to use style sheets from an unconfigured project, such as Module parse failed: Unexpected character '@' (14:0).

You can use cssSupport.mode to specify how encountered CSS should be handled. The following values are available:

- "inline" (default): styles are added to the global <style> tag.
- "extract": styles are extracted into a separate file. They can then be included from an HTML page.
- "import": styles are processed as strings. This can be useful if you need access to the CSS from your code (such as val styles = require("main.css")).

To use different modes for the same project, use cssSupport.rules. Here, you can specify a list of KotlinWebpackCssRules, each of which define a mode, as well as include and exclude patterns.

Node.js

For Kotlin/JS projects targeting Node.js, the plugin automatically downloads and installs the Node.js environment on the host. You can also use an existing Node.js instance if you have it.

Use pre-installed Node.js

If Node.js is already installed on the host where you build Kotlin/JS projects, you can configure the Kotlin/JS Gradle plugin to use it instead of installing its own Node.js instance.

To use the pre-installed Node.js instance, add the following lines to your build.gradle(.kts):

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension>().download = false
    // or true for default behavior
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension).download = false
}
```

Yarn

To download and install your declared dependencies at build time, the plugin manages its own instance of the Yarn package manager. It works out of the box without additional configuration, but you can tune it or use Yarn already installed on your host.

Additional Yarn features: .yarnrc

To configure additional Yarn features, place a .yarnrc file in the root of your project. At build time, it gets picked up automatically.

For example, to use a custom registry for npm packages, add the following line to a file called .yarnrc in the project root:

```
registry "http://my.registry/api/npm/"
```

To learn more about .yarnrc, please visit the official Yarn documentation.

Use pre-installed Yarn

If Yarn is already installed on the host where you build Kotlin/JS projects, you can configure the Kotlin/JS Gradle plugin to use it instead of installing its own Yarn instance.

To use the pre-installed Yarn instance, add the following lines to your build.gradle(.kts):

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().download = false
    // or true for default behavior
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).download = false
}
```

Version locking via kotlin-js-store

Version locking via kotlin-js-store is available since Kotlin 1.6.10.

The kotlin-js-store directory in the project root is automatically generated by the Kotlin Multiplatform Gradle plugin to hold the yarn.lock file, which is necessary for version locking. The lockfile is entirely managed by the Yarn plugin and gets updated during the execution of the kotlinNpmInstall Gradle task.

To follow a recommended practice, commit kotlin-js-store and its contents to your version control system. It ensures that your application is being built with the exact same dependency tree on all machines.

If needed, you can change both directory and lockfile names in the build script:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().lockFileDirectory =
        project.rootDir.resolve("my-kotlin-js-store")
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().lockFileName = "my-yarn.lock"
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFileDirectory =
        file("my-kotlin-js-store")
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFileName = 'my-yarn.lock'
}
```

Changing the name of the lockfile may cause dependency inspection tools to no longer pick up the file.

To learn more about yarn.lock, please visit the official Yarn documentation.

Reporting that yarn.lock has been updated

Kotlin/JS provides Gradle settings that could notify you if the yarn.lock file has been updated. You can use these settings when you want to be notified if yarn.lock has been changed silently during the CI build process:

- YarnLockMismatchReport, which specifies how changes to the yarn.lock file are reported. You can use one of the following values:
  - FAIL fails the corresponding Gradle task. This is the default.
  - WARNING writes the information about changes in the warning log.
  - NONE disables reporting.
- reportNewYarnLock, which reports about the recently created yarn.lock file explicitly. By default, this option is disabled: it's a common practice to generate a new yarn.lock file at the first start. You can use this option to ensure that the file has been committed to your repository.
- yarnLockAutoReplace, which replaces yarn.lock automatically every time the Gradle task is run.

To use these options, update your build script file build.gradle(.kts) as follows:

Kotlin

```
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin::class.java) {
    rootProject.the<YarnRootExtension>().yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.the<YarnRootExtension>().reportNewYarnLock = false // true
    rootProject.the<YarnRootExtension>().yarnLockAutoReplace = false // true
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).reportNewYarnLock = false //
true
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).yarnLockAutoReplace = false //
true
}
```

Installing npm dependencies with --ignore-scripts by default

> Installing npm dependencies with --ignore-scripts by default is available since Kotlin 1.6.10.

To reduce the likelihood of executing malicious code from compromised npm packages, the Kotlin Multiplatform Gradle plugin prevents the execution of lifecycle scripts during the installation of npm dependencies by default.

You can explicitly enable lifecycle scripts execution by adding the following lines to build.gradle(.kts):

Kotlin

```kotlin
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().ignoreScripts = false
}
```

Groovy

```groovy
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).ignoreScripts = false
}
```

Distribution target directory

By default, the results of a Kotlin/JS project build reside in the /build/dist/<targetName>/<binaryName> directory within the project root.

> Prior to Kotlin 1.9.0, the default distribution target directory was /build/distributions.

To set another location for project distribution files, add the distribution block inside browser in the build script and assign a value to the directory property. Once you run a project build task, Gradle will save the output bundle in this location together with project resources.

Kotlin

```kotlin
kotlin {
    js {
        browser {
            distribution {
                directory = File("$projectDir/output/")
            }
        }
        binaries.executable()
        // ...
    }
}
```

Groovy

```
kotlin {
    js {
        browser {
            distribution {
                directory = file("$projectDir/output/")
            }
        }
        binaries.executable()
        // ...
    }
}
```

Module name

To adjust the name for the JavaScript module (which is generated in build/js/packages/myModuleName), including the corresponding .js and .d.ts files, use the moduleName option:

```
js {
    moduleName = "myModuleName"
}
```

Note that this does not affect the webpacked output in build/dist.

package.json customization

The package.json file holds the metadata of a JavaScript package. Popular package registries such as npm require all published packages to have such a file. They use it to track and manage package publications.

The Kotlin Multiplatform Gradle plugin automatically generates package.json for Kotlin/JS projects during build time. By default, the file contains essential data: name, version, license, and dependencies, and some other package attributes.

Aside from basic package attributes, package.json can define how a JavaScript project should behave, for example, identifying scripts that are available to run.

You can add custom entries to the project's package.json via the Gradle DSL. To add custom fields to your package.json, use the customField function in the compilations packageJson block:

```
kotlin {
    js {
        compilations["main"].packageJson {
            customField("hello", mapOf("one" to 1, "two" to 2))
        }
    }
}
```

When you build the project, this code will add the following block to the package.json file:

```
"hello": {
    "one": 1,
    "two": 2
}
```

Learn more about writing package.json files for npm registry in the npm docs.

Troubleshooting

When building a Kotlin/JS project using Kotlin 1.3.xx, you may encounter a Gradle error if one of your dependencies (or any transitive dependency) was built using Kotlin 1.4 or higher: Could not determine the dependencies of task ':client:jsTestPackageJson'. / Cannot choose between the following variants. This is a known problem, a workaround is provided here.

Run Kotlin/JS

Since Kotlin/JS projects are managed with the Kotlin Multiplatform Gradle plugin, you can run your project using the appropriate tasks. If you're starting with a blank project, ensure that you have some sample code to execute. Create the file src/jsMain/kotlin/App.kt and fill it with a small "Hello, World"-type code snippet:

```kotlin
fun main() {
    console.log("Hello, Kotlin/JS!")
}
```

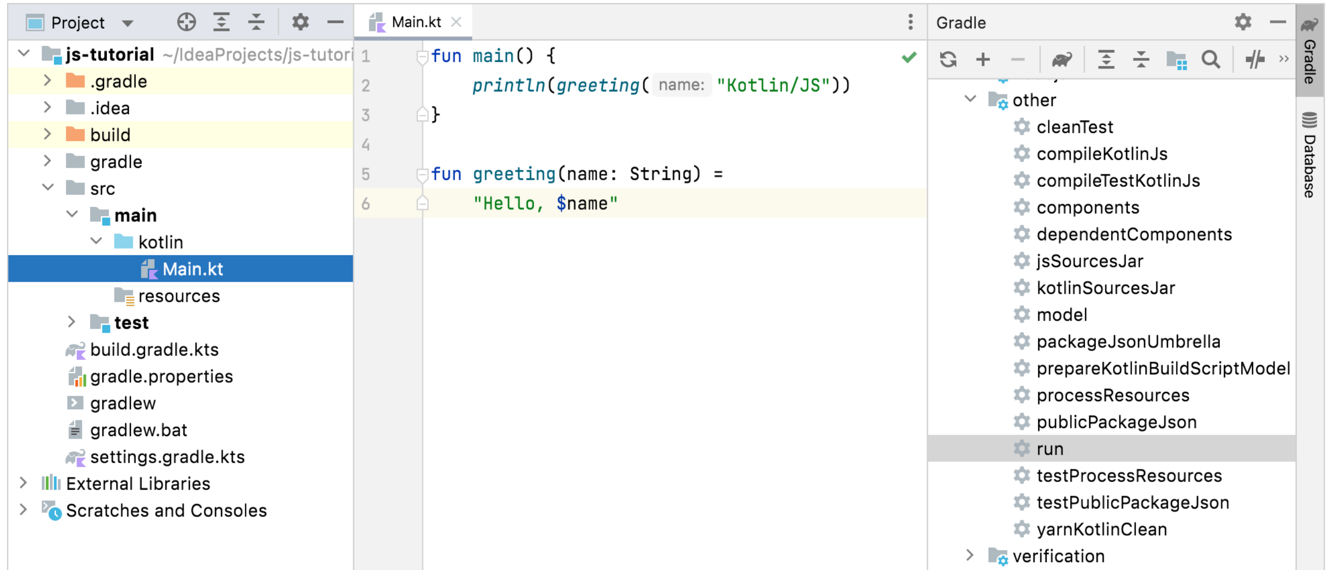
Depending on the target platform, some platform-specific extra setup might be required to run your code for the first time.

Run the Node.js target

When targeting Node.js with Kotlin/JS, you can simply execute the jsRun Gradle task. This can be done for example via the command line, using the Gradle wrapper:

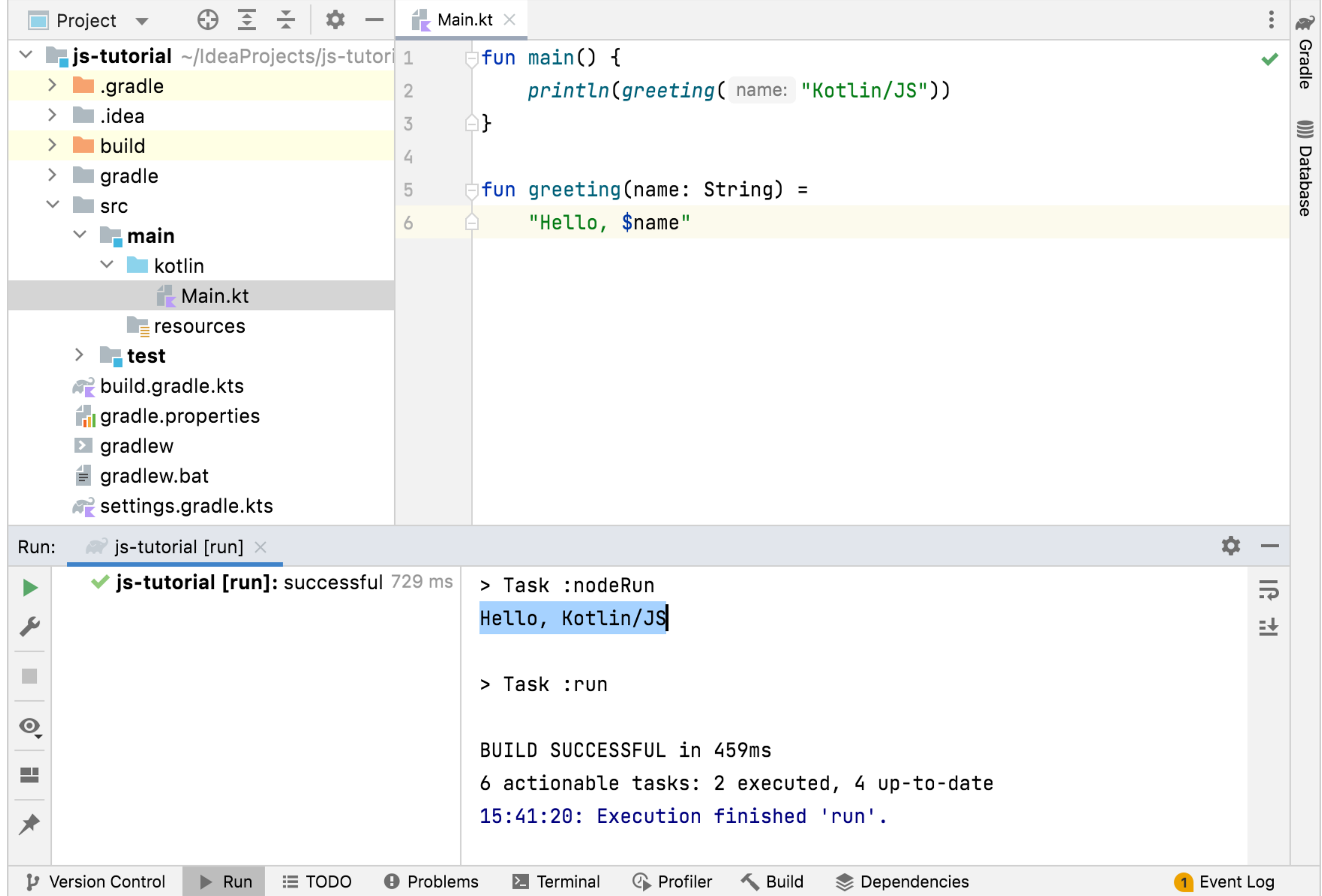
```
./gradlew jsRun
```

If you're using IntelliJ IDEA, you can find the jsRun action in the Gradle tool window:



Gradle Run task in IntelliJ IDEA

On first start, the kotlin.multiplatform Gradle plugin will download all required dependencies to get you up and running. After the build is completed, the program is executed, and you can see the logging output in the terminal:



Executing the JS target in a Kotlin Multiplatform project in IntelliJ IDEA

Run the browser target

When targeting the browser, your project is required to have an HTML page. This page will be served by the development server while you are working on your application, and should embed your compiled Kotlin/JS file. Create and fill an HTML file /src/jsMain/resources/index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JS Client</title>
</head>
<body>
<script src="js-tutorial.js"></script>
</body>
</html>
```

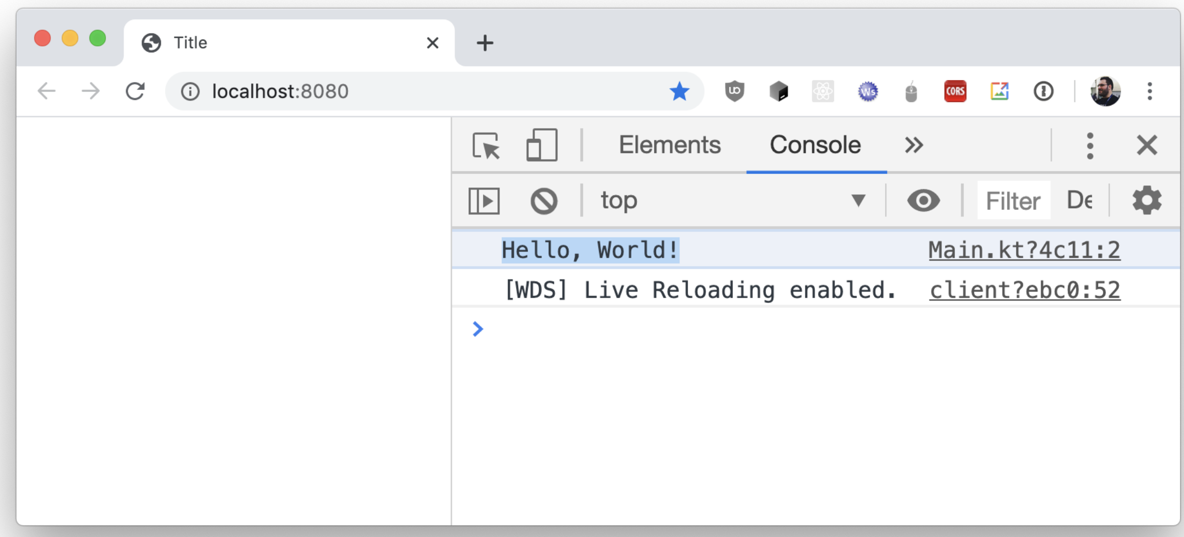
By default, the name of your project's generated artifact (which is created through webpack) that needs to be referenced is your project name (in this case, js-tutorial). If you've named your project followAlong, make sure to embed followAlong.js instead of js-tutorial.js

After making these adjustments, start the integrated development server. You can do this from the command line via the Gradle wrapper:

```
./gradlew jsRun
```

When working from IntelliJ IDEA, you can find the jsRun action in the Gradle tool window.

After the project has been built, the embedded webpack-dev-server will start running, and will open a (seemingly empty) browser window pointing to the HTML file you specified previously. To validate that your program is running correctly, open the developer tools of your browser (for example by right-clicking and choosing the Inspect action). Inside the developer tools, navigate to the console, where you can see the results of the executed JavaScript code:



Console output in browser developer tools

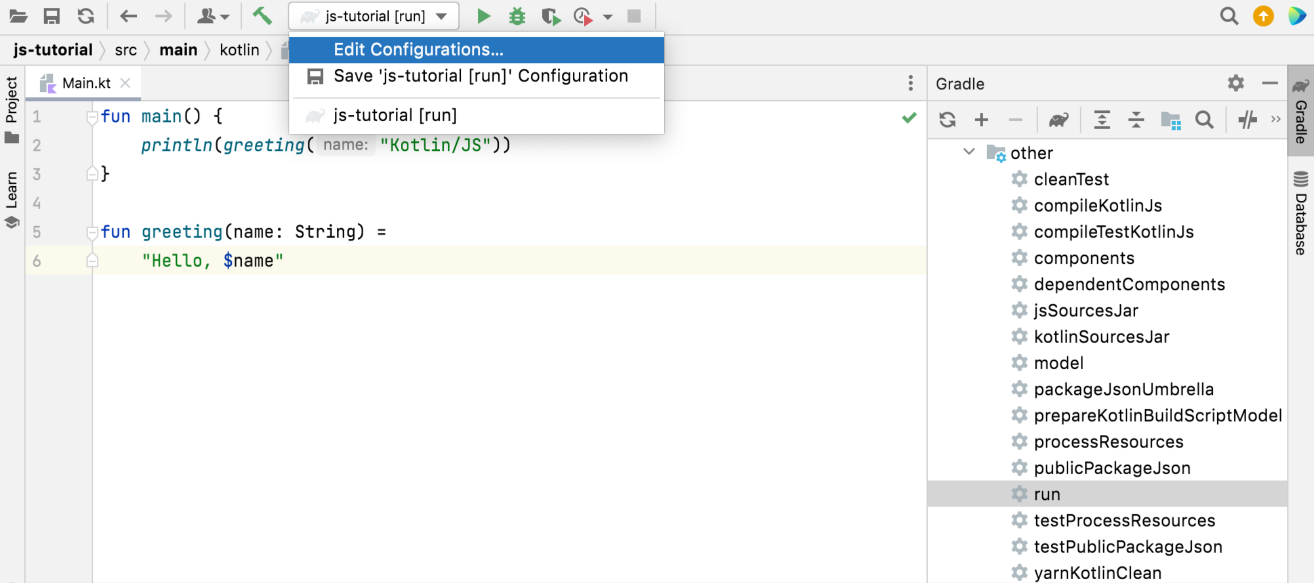
With this setup, you can recompile your project after each code change to see your changes. Kotlin/JS also supports a more convenient way of automatically rebuilding the application while you are developing it. To find out how to set up this continuous mode, check out the corresponding tutorial.

Development server and continuous compilation

Instead of manually compiling and executing a Kotlin/JS project every time you want to see the changes you made, you can use the continuous compilation mode. Instead of using the regular run command, invoke the Gradle wrapper in continuous mode:

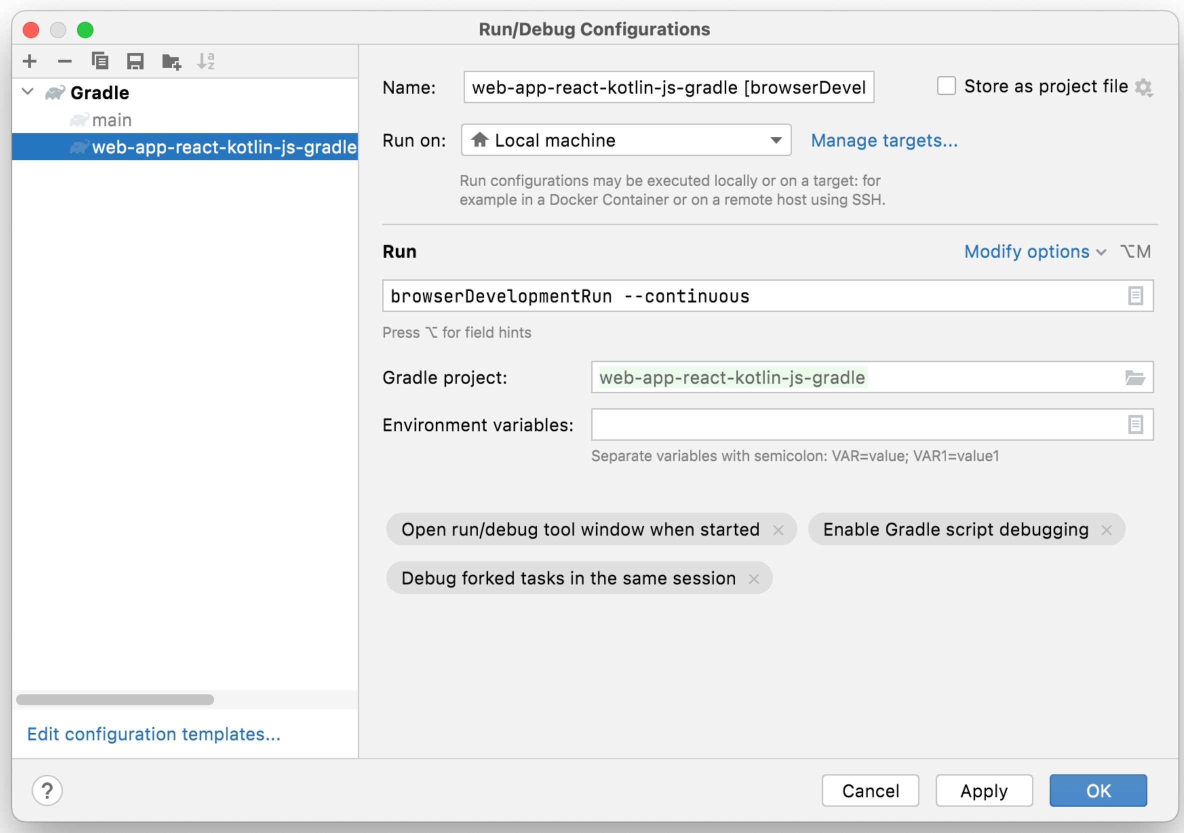
```
./gradlew run --continuous
```

If you are working in IntelliJ IDEA, you can pass the same flag via the run configuration. After running the Gradle run task for the first time from the IDE, IntelliJ IDEA automatically generates a run configuration for it, which you can edit:



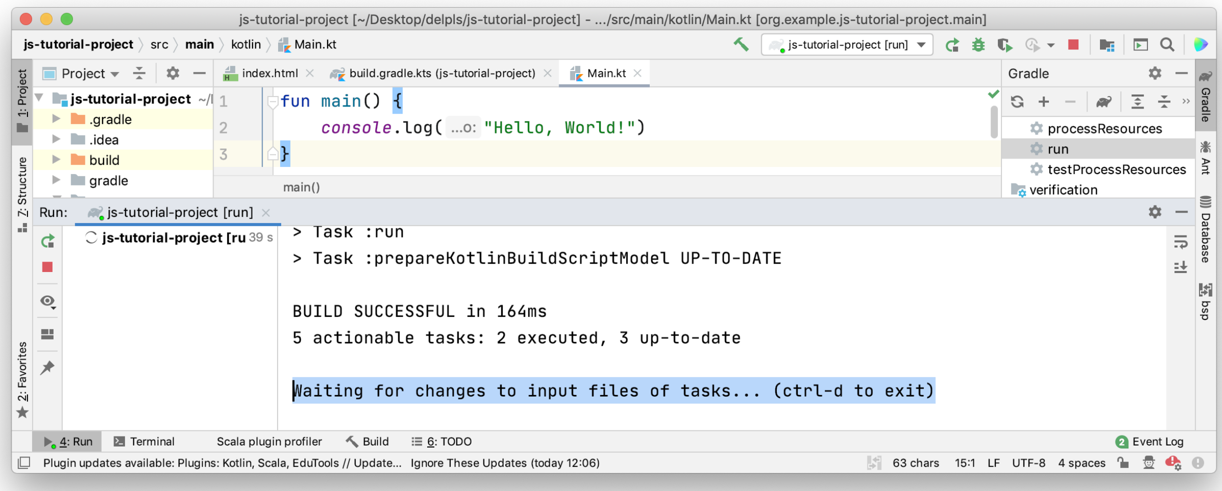
Editing run configurations in IntelliJ IDEA

Enabling continuous mode via the Run/Debug Configurations dialog is as easy as adding the --continuous flag to the arguments for the run configuration:



Adding the continuous flag to a run configuration in IntelliJ IDEA

When executing this run configuration, you can note that the Gradle process continues watching for changes to the program:



Gradle waiting for changes

Once a change has been detected, the program will be recompiled automatically. If you still have the page open in the browser, the development server will trigger an automatic reload of the page, and the changes will become visible. This is thanks to the integrated webpack-dev-server that is managed by the Kotlin Multiplatform Gradle plugin.

Debug Kotlin/JS code

JavaScript source maps provide mappings between the minified code produced by bundlers or minifiers and the actual source code a developer works with. This way, the source maps enable support for debugging the code during its execution.

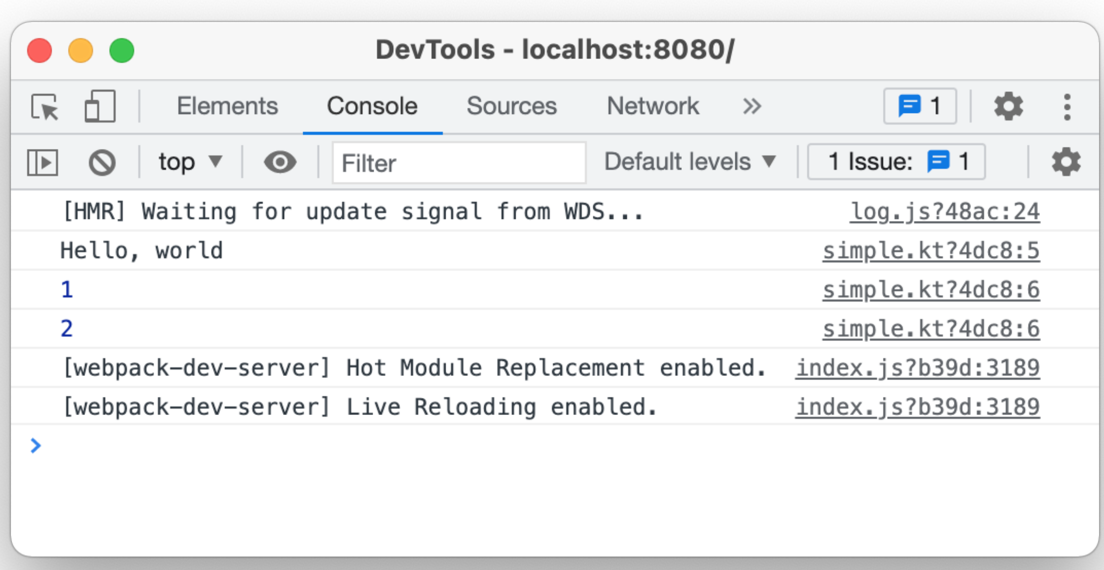
The Kotlin Multiplatform Gradle plugin automatically generates source maps for the project builds, making them available without any additional configuration.

Debug in browser

Most modern browsers provide tools that allow inspecting the page content and debugging the code that executes on it. Refer to your browser's documentation for more details.

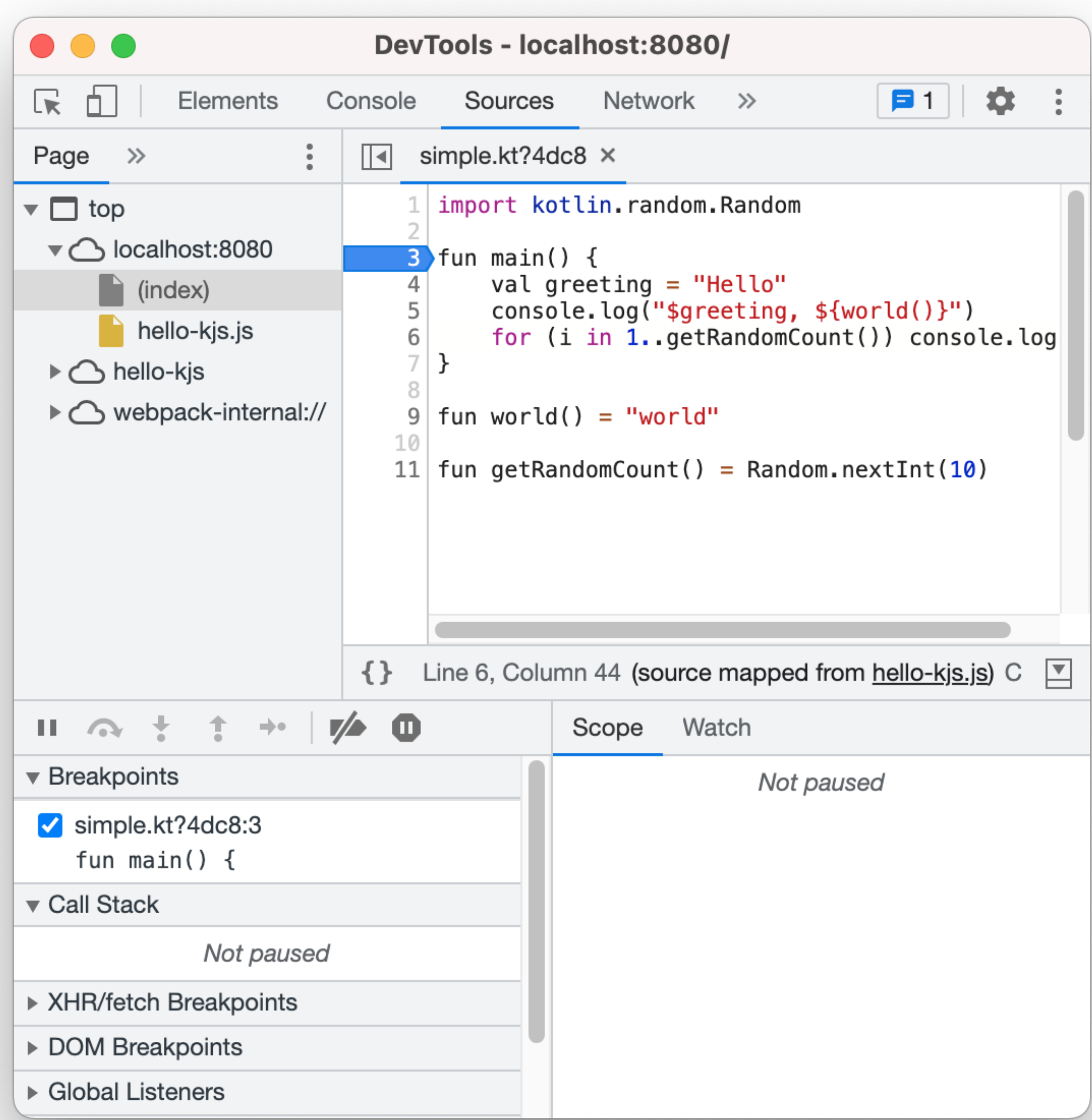
To debug Kotlin/JS in the browser:

1. Run the project by calling one of the available run Gradle tasks, for example, browserDevelopmentRun or jsBrowserDevelopmentRun in a multiplatform project. Learn more about running Kotlin/JS.
2. Navigate to the page in the browser and launch its developer tools (for example, by right-clicking and selecting the Inspect action). Learn how to find the developer tools in popular browsers.
3. If your program is logging information to the console, navigate to the Console tab to see this output. Depending on your browser, these logs can reference the Kotlin source files and lines they come from:



Chrome DevTools console

4. Click the file reference on the right to navigate to the corresponding line of code. Alternatively, you can manually switch to the Sources tab and find the file you need in the file tree. Navigating to the Kotlin file shows you the regular Kotlin code (as opposed to minified JavaScript):



Debugging in Chrome DevTools

You can now start debugging the program. Set a breakpoint by clicking on one of the line numbers. The developer tools even support setting breakpoints within a statement. As with regular JavaScript code, any set breakpoints will persist across page reloads. This also makes it possible to debug Kotlin's main() method which is executed when the script is loaded for the first time.

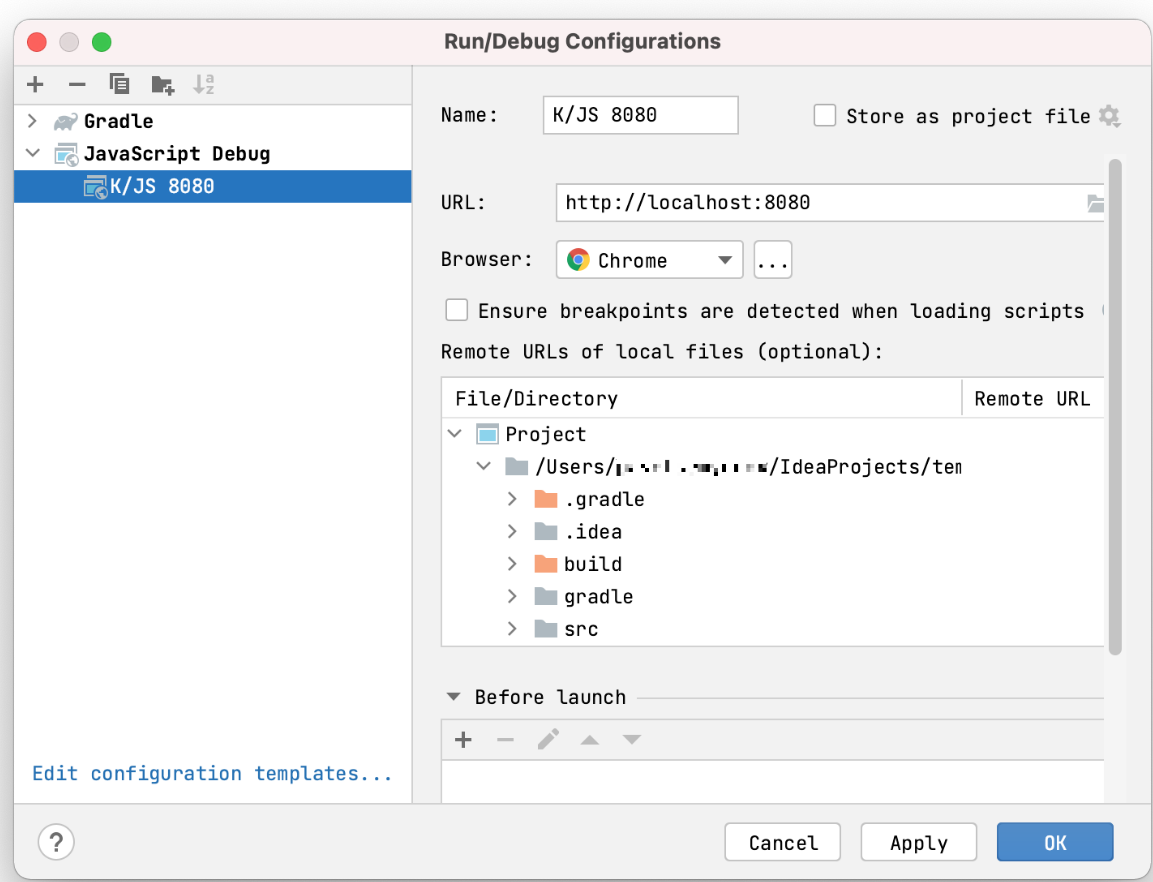
Debug in the IDE

IntelliJ IDEA Ultimate provides a powerful set of tools for debugging code during development.

For debugging Kotlin/JS in IntelliJ IDEA, you'll need a JavaScript Debug configuration. To add such a debug configuration:

1. Go to Run | Edit Configurations.
2. Click + and select JavaScript Debug.

3. Specify the configuration Name and provide the URL on which the project runs (http://localhost:8080 by default).



JavaScript debug configuration

4. Save the configuration.

Learn more about setting up JavaScript debug configurations.

Now you're ready to debug your project!

1. Run the project by calling one of the available run Gradle tasks, for example, browserDevelopmentRun or jsBrowserDevelopmentRun in a multiplatform project. Learn more about running Kotlin/JS.

2. Start the debugging session by running the JavaScript debug configuration you've created previously:

hello-kjs – simple.kt [hello-kjs.main]

getRandomCount() K/JS 8080

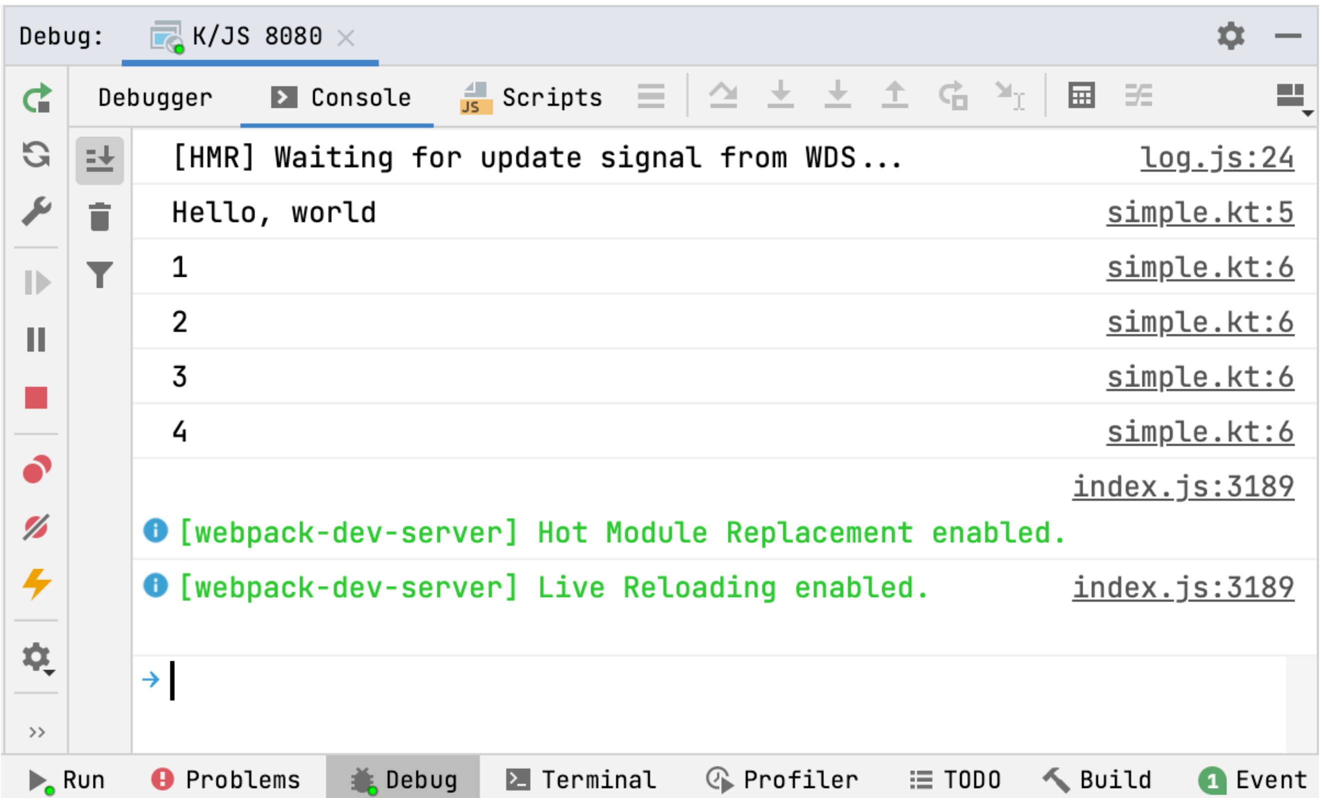
simple.kt index.html favicon.ico

Debug 'K/JS 8080'

```kotlin
import kotlin.random.Random

fun main() {
    val greeting = "Hello"
    console.log( ...o: "$greeting, ${world()}")
    for (i in 1..getRandomCount()) console.log(i)
}

fun world() = "world"

fun getRandomCount() = Random.nextInt( until: 10)
```

JavaScript debug configuration

3. You can see the console output of your program in the Debug window in IntelliJ IDEA. The output items reference the Kotlin source files and lines they come from:



JavaScript debug output in the IDE

4. Click the file reference on the right to navigate to the corresponding line of code.

You can now start debugging the program using the whole set of tools that the IDE offers: breakpoints, stepping, expression evaluation, and more. Learn more about debugging in IntelliJ IDEA.

> Because of the limitations of the current JavaScript debugger in IntelliJ IDEA, you may need to rerun the JavaScript debug to make the execution stop on breakpoints.

Debug in Node.js

If your project targets Node.js, you can debug it in this runtime.

To debug a Kotlin/JS application targeting Node.js:

1. Build the project by running the build Gradle task.
2. Find the resulting .js file for Node.js in the build/js/packages/your-module/kotlin/ directory inside your project's directory.
3. Debug it in Node.js as described in the Node.js Debugging Guide.

What's next?

Now that you know how to start debug sessions with your Kotlin/JS project, learn to make efficient use of the debugging tools:

- Learn how to debug JavaScript in Google Chrome
- Get familiar with IntelliJ IDEA JavaScript debugger
- Learn how to debug in Node.js.

If you run into any problems

If you face any issues with debugging Kotlin/JS, please report them to our issue tracker, YouTrack

Run tests in Kotlin/JS

The Kotlin Multiplatform Gradle plugin lets you run tests through a variety of test runners that can be specified via the Gradle configuration.

When you create a multiplatform project, the Project Wizard automatically adds test dependencies to all the source sets. If you created your project without it, you can add the dependencies manually to make test annotations and functionality available for the JavaScript target:

Kotlin

```
// build.gradle.kts

kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // This brings all the platform dependencies automatically
            }
        }
    }
}
```

Groovy

```
// build.gradle

kotlin {
    sourceSets {
```

```
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings all the platform dependencies automatically
            }
        }
    }
}
```

You can tune how tests are executed in Kotlin/JS by adjusting the settings available in the testTask block in the Gradle build script. For example, using the Karma test runner together with a headless instance of Chrome and an instance of Firefox looks like this:

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useChromeHeadless()
                    useFirefox()
                }
            }
        }
    }
}
```

For a detailed description of the available functionality, check out the Kotlin/JS reference on configuring the test task.

Please note that by default, no browsers are bundled with the plugin. This means that you'll have to ensure they're available on the target system.

To check that tests are executed properly, add a file src/jsTest/kotlin/AppTest.kt and fill it with this content:

```
import kotlin.test.Test
import kotlin.test.assertEquals

class AppTest {
    @Test
    fun thingsShouldWork() {
        assertEquals(listOf(1,2,3).reversed(), listOf(3,2,1))
    }

    @Test
    fun thingsShouldBreak() {
        assertEquals(listOf(1,2,3).reversed(), listOf(1,2,3))
    }
}
```

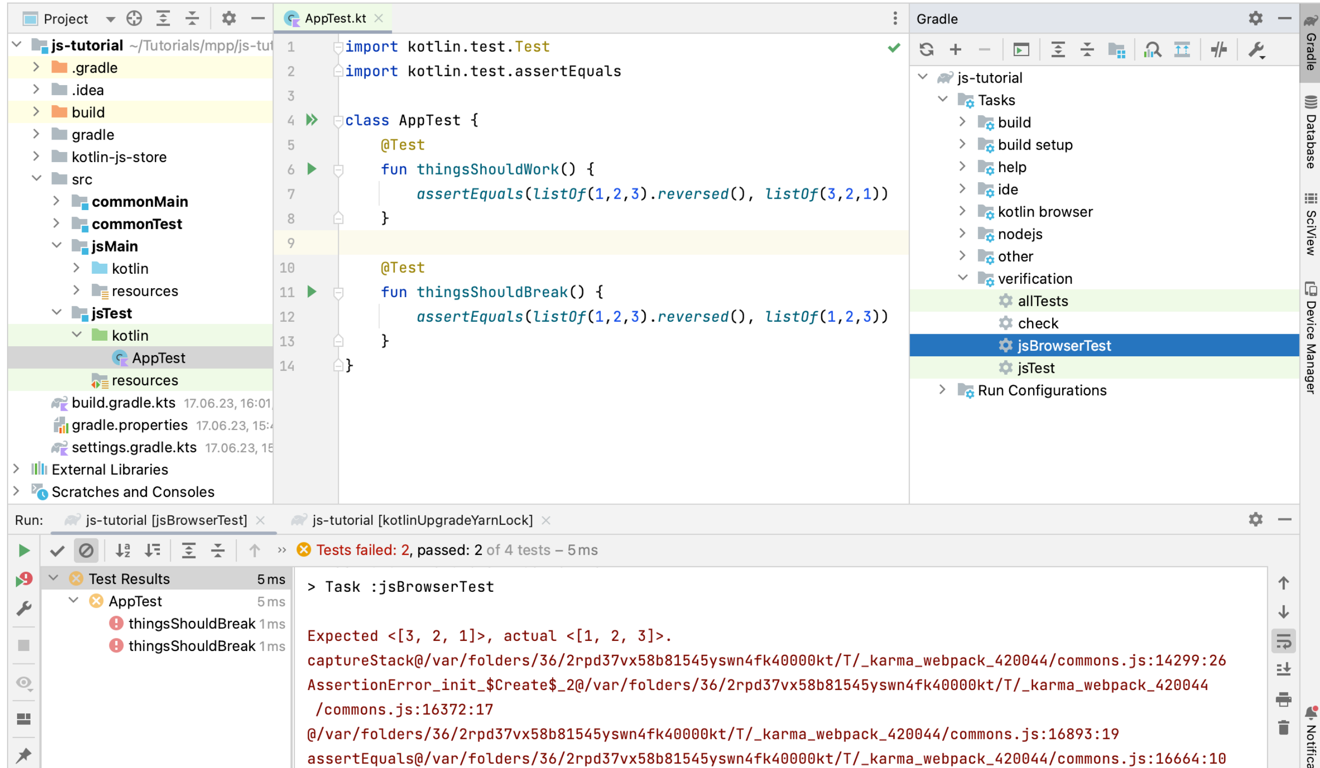
To run the tests in the browser, execute the jsBrowserTest task via IntelliJ IDEA, or use the gutter icons to execute all or individual tests:

Gradle browserTest task

Alternatively, if you want to run the tests via the command line, use the Gradle wrapper:

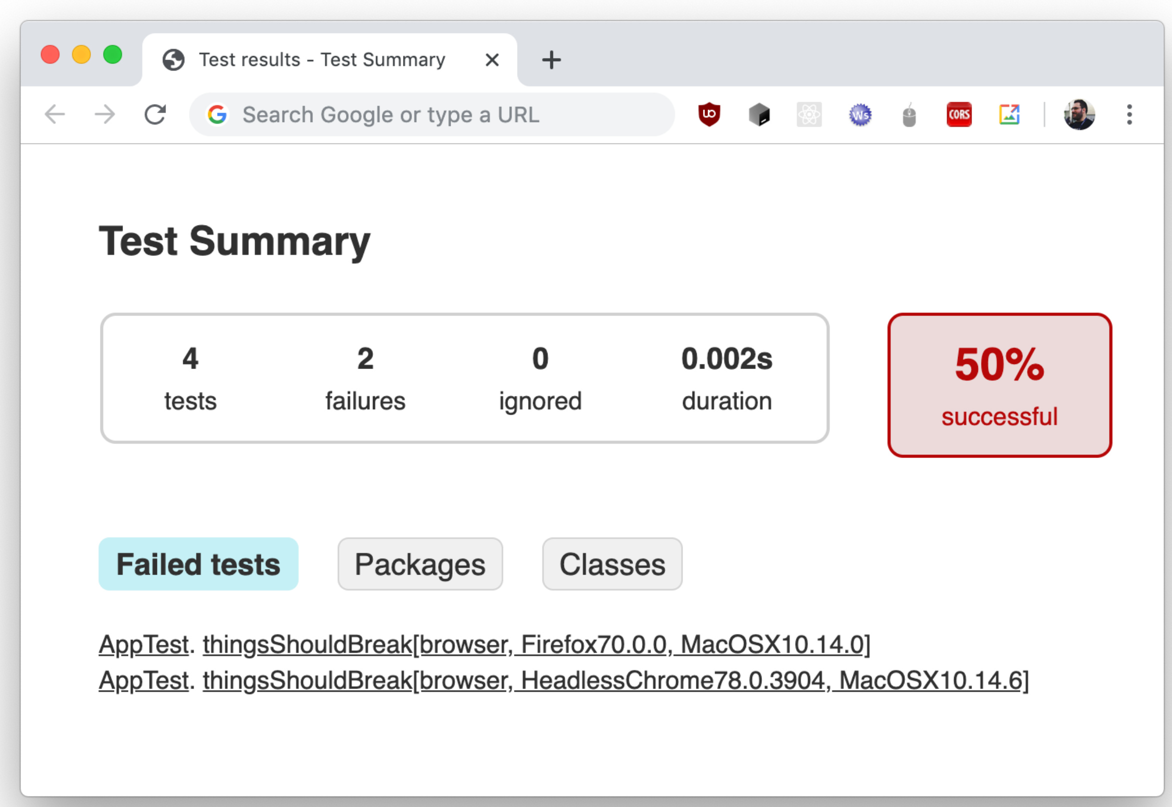
```
./gradlew jsBrowserTest
```

After running the tests from IntelliJ IDEA, the Run tool window will show the test results. You can click failed tests to see their stack trace, and navigate to the corresponding test implementation via a double click.



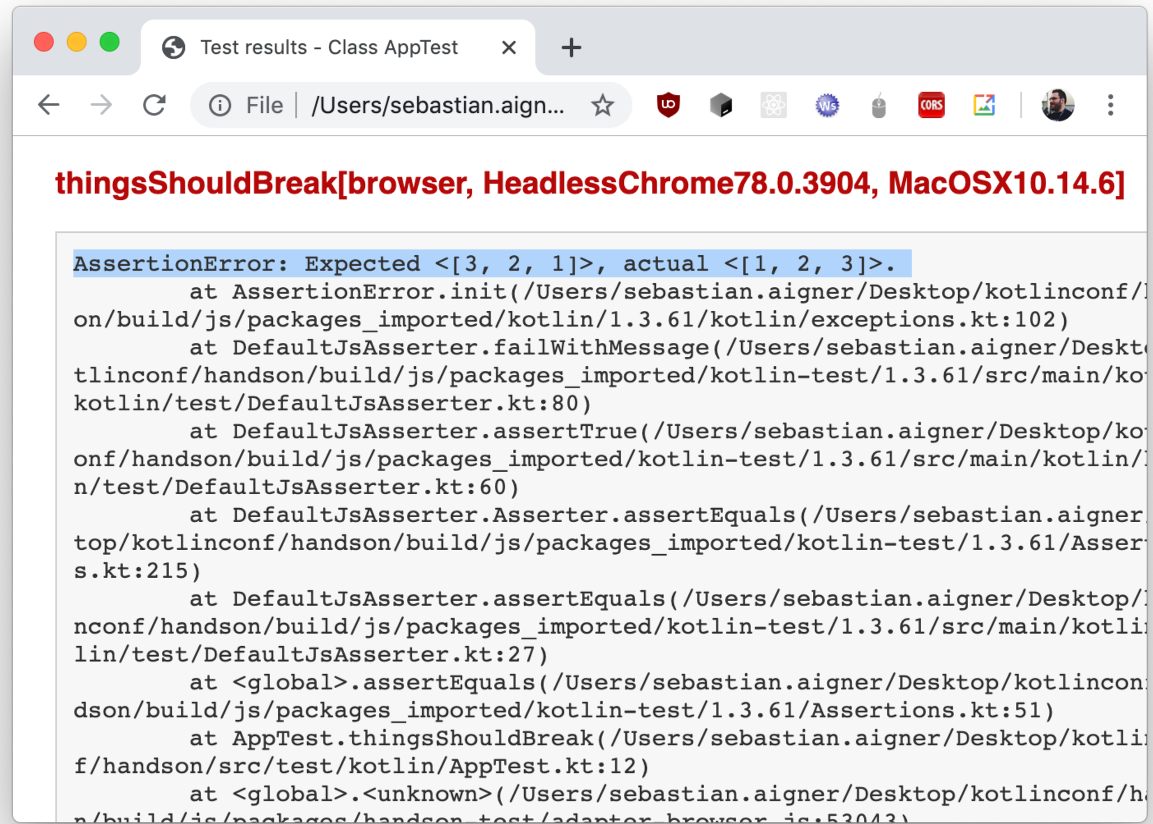
Test results in IntelliJ IDEA

After each test run, regardless of how you executed the test, you can find a properly formatted test report from Gradle in build/reports/tests/jsBrowserTest/index.html. Open this file in a browser to see another overview of the test results:



Gradle test summary

If you are using the set of example tests shown in the snippet above, one test passes, and one test breaks, which gives the resulting total of 50% successful tests. To get more information about individual test cases, you can navigate via the provided hyperlinks:



Stacktrace of failed test in the Gradle summary

Kotlin/JS dead code elimination

The Kotlin Multiplatform Gradle plugin includes a dead code elimination (DCE) tool. Dead code elimination is often also called tree shaking. It reduces the size or the resulting JavaScript code by removing unused properties, functions, and classes.

Unused declarations can appear in cases like:

- A function is inlined and never gets called directly (which happens always except for a few situations).
- A module uses a shared library. Without DCE, parts of the library that you don't use are still included in the resulting bundle. For example, the Kotlin standard library contains functions for manipulating lists, arrays, char sequences, adapters for DOM, and so on. All of this functionality would require about 1.3 MB as a JavaScript file. A simple "Hello, world" application only requires console routines, which is only few kilobytes for the entire file.

The Kotlin Multiplatform Gradle plugin handles DCE automatically when you build a production bundle, for example by using the browserProductionWebpack task. Development bundling tasks (like browserDevelopmentWebpack) don't include DCE.

Exclude declarations from DCE

Sometimes you may need to keep a function or a class in the resulting JavaScript code even if you don't use it in your module, for example, if you're going to use it in the client JavaScript code.

To keep certain declarations from elimination, add the dceTask block to your Gradle build script and list the declarations as arguments of the keep function. An argument must be the declaration's fully qualified name with the module name as a prefix: moduleName.dot.separated.package.name.declarationName

> Unless specified otherwise, the names of functions and modules can be mangled in the generated JavaScript code. To keep such functions from elimination, use the mangled names in the keep arguments as they appear in the generated JavaScript code.

```
kotlin {
    js {
        browser {
            dceTask {
                keep("myKotlinJSModule.org.example.getName", "myKotlinJSModule.org.example.User" )
            }
            binaries.executable()
        }
    }
}
```

If you want to keep a whole package or module from elimination, you can use its fully qualified name as it appears in the generated JavaScript code.

> Keeping whole packages or modules from elimination can prevent DCE from removing many unused declarations. Because of this, it is preferable to select individual declarations which should be excluded from DCE one by one.

Disable DCE

To turn off DCE completely, use the devMode option in the dceTask:

```
kotlin {
    js {
        browser {
            dceTask {
                dceOptions.devMode = true
            }
        }
        binaries.executable()
    }
}
```

Kotlin/JS IR compiler

The Kotlin/JS IR compiler backend is the main focus of innovation around Kotlin/JS, and paves the way forward for the technology.

Rather than directly generating JavaScript code from Kotlin source code, the Kotlin/JS IR compiler backend leverages a new approach. Kotlin source code is first transformed into a Kotlin intermediate representation (IR), which is subsequently compiled into JavaScript. For Kotlin/JS, this enables aggressive optimizations, and allows improvements on pain points that were present in the previous compiler, such as generated code size (through dead code elimination), and JavaScript and TypeScript ecosystem interoperability, to name some examples.

The IR compiler backend is available starting with Kotlin 1.4.0 through the Kotlin Multiplatform Gradle plugin. To enable it in your project, pass a compiler type to the js function in your Gradle build script:

```
kotlin {
    js(IR) { // or: LEGACY, BOTH
        // ...
        binaries.executable() // not applicable to BOTH, see details below
    }
}
```

- IR uses the new IR compiler backend for Kotlin/JS.
- LEGACY uses the default compiler backend.
- BOTH compiles your project with the new IR compiler as well as the default compiler backend. Use this mode for authoring libraries compatible with both backends.

The compiler type can also be set in the gradle.properties file, with the key kotlin.js.compiler=ir. This behaviour is overwritten by any settings in the build.gradle(.kts), however.

Lazy initialization of top-level properties

For better application startup performance, the Kotlin/JS IR compiler initializes top-level properties lazily. This way, the application loads without initializing all the top-level properties used in its code. It initializes only the ones needed at startup; other properties receive their values later when the code that uses them actually runs.

```
val a = run {
    val result = // intensive computations
    println(result)
    result
} // value is computed upon the first usage
```

If for some reason you need to initialize a property eagerly (upon the application start), mark it with the @EagerInitialization annotation.

Incremental compilation for development binaries

The JS IR compiler provides the incremental compilation mode for development binaries that speeds up the development process. In this mode, the compiler caches the results of compileDevelopmentExecutableKotlinJs Gradle task on the module level. It uses the cached compilation results for unchanged source files during subsequent compilations, making them complete faster, especially with small changes.

Incremental compilation is enabled by default. To disable incremental compilation for development binaries, add the following line to the project's gradle.properties or local.properties:

```
kotlin.incremental.js.ir=false // true by default
```

> The clean build in the incremental compilation mode is usually slower because of the need to create and populate the caches.

Output .js files: one per module or one for the whole project

As a compilation result, the JS IR compiler outputs separate .js files for each module of a project. Alternatively, you can compile the whole project into a single .js file by adding the following line to gradle.properties:

```
kotlin.js.ir.output.granularity=whole-program // 'per-module' is the default
```

Ignoring compilation errors

> Ignore compilation errors mode is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin/JS IR compiler provides a new compilation mode unavailable in the default backend – ignoring compilation errors. In this mode, you can try out your application even while its code contains errors. For example, when you're doing a complex refactoring or working on a part of the system that is completely unrelated to a compilation error in another part.

With this new compiler mode, the compiler ignores all broken code. Thus, you can run the application and try its parts that don't use the broken code. If you try to run the code that was broken during compilation, you'll get a runtime exception.

Choose between two tolerance policies for ignoring compilation errors in your code:

- SEMANTIC. The compiler will accept code that is syntactically correct but doesn't make sense semantically. For example, assigning a number to a string variable (type mismatch).
- SYNTAX. The compiler will accept any code, even if it contains syntax errors. Regardless of what you write, the compiler will still try to generate a runnable executable.

As an experimental feature, ignoring compilation errors requires an opt-in. To enable this mode, add the -Xerror-tolerance-policy={SEMANTIC|SYNTAX} compiler option:

```kotlin
kotlin {
    js(IR) {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions.freeCompilerArgs.add("-Xerror-tolerance-policy=SYNTAX")
            }
        }
    }
}
```

Minification of member names in production

The Kotlin/JS IR compiler uses its internal information about the relationships of your Kotlin classes and functions to apply more efficient minification, shortening the names of functions, properties, and classes. This reduces the size of resulting bundled applications.

This type of minification is automatically applied when you build your Kotlin/JS application in production mode, and enabled by default. To disable member name minification, use the -Xir-minimized-member-names compiler option:

```kotlin
kotlin {
    js(IR) {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions.freeCompilerArgs.add("-Xir-minimized-member-names=false")
            }
        }
    }
}
```

Preview: generation of TypeScript declaration files (d.ts)

> The generation of TypeScript declaration files (d.ts) is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The Kotlin/JS IR compiler is capable of generating TypeScript definitions from your Kotlin code. These definitions can be used by JavaScript tools and IDEs when working on hybrid apps to provide autocompletion, support static analyzers, and make it easier to include Kotlin code in JavaScript and TypeScript projects.

If your project produces executable files (binaries.executable()), the Kotlin/JS IR compiler collects any top-level declarations marked with @JsExport and automatically generates TypeScript definitions in a .d.ts file.

If you want to generate TypeScript definitions, you have to explicitly configure this in your Gradle build file. Add generateTypeScriptDefinitions() to your build.gradle.kts file in the js section. For example:

```kotlin
kotlin {
   js {
       binaries.executable()
       browser {
       }
       generateTypeScriptDefinitions()
   }
}
```

The definitions can be found in build/js/packages/<package_name>/kotlin alongside the corresponding un-webpacked JavaScript code.

Current limitations of the IR compiler

A major change with the new IR compiler backend is the absence of binary compatibility with the default backend. A library created with the new IR compiler uses a klib format and can't be used from the default backend. In the meantime, a library created with the old compiler is a jar with js files, which can't be used from the IR backend.

If you want to use the IR compiler backend for your project, you need to update all Kotlin dependencies to versions that support this new backend. Libraries published by JetBrains for Kotlin 1.4+ targeting Kotlin/JS already contain all artifacts required for usage with the new IR compiler backend.

If you are a library author looking to provide compatibility with the current compiler backend as well as the new IR compiler backend, additionally check out the section about authoring libraries for the IR compiler section.

The IR compiler backend also has some discrepancies in comparison to the default backend. When trying out the new backend, it's good to be mindful of these possible pitfalls.

- Some libraries that rely on specific characteristics of the default backend, such as kotlin-wrappers, can display some problems. You can follow the investigation and progress on YouTrack.
- The IR backend does not make Kotlin declarations available to JavaScript by default at all. To make Kotlin declarations visible to JavaScript, they must be annotated with @JsExport.

Migrating existing projects to the IR compiler

Due to significant differences between the two Kotlin/JS compilers, making your Kotlin/JS code work with the IR compiler may require some adjustments. Learn how to migrate existing Kotlin/JS projects to the IR compiler in the Kotlin/JS IR compiler migration guide.

Authoring libraries for the IR compiler with backwards compatibility

If you're a library maintainer who is looking to provide compatibility with the default backend as well as the new IR compiler backend, a setting for the compiler selection is available that allows you to create artifacts for both backends, allowing you to keep compatibility for your existing users while providing support for the next generation of Kotlin compiler. This so-called both-mode can be turned on using the kotlin.js.compiler=both setting in your gradle.properties file, or can be set as one of the project-specific options inside your js block inside the build.gradle(.kts) file:

```
kotlin {
    js(BOTH) {
        // ...
    }
}
```

When in both mode, the IR compiler backend and default compiler backend are both used when building a library from your sources (hence the name). This means that both klib files with Kotlin IR as well as jar files for the default compiler will be generated. When published under the same Maven coordinate, Gradle will automatically choose the right artifact depending on the use case – js for the old compiler, klib for the new one. This enables you to compile and publish your library for projects that are using either of the two compiler backends.

Migrating Kotlin/JS projects to the IR compiler

We replaced the old Kotlin/JS compiler with the IR-based compiler in order to unify Kotlin's behavior on all platforms and to make it possible to implement new JS-specific optimizations, among other reasons. You can learn more about the internal differences between the two compilers in the blog post Migrating our Kotlin/JS app to the new IR compiler by Sebastian Aigner.

Due to the significant differences between the compilers, switching your Kotlin/JS project from the old backend to the new one may require adjusting your code. On this page, we've compiled a list of known migration issues along with suggested solutions.

> Install the Kotlin/JS Inspection pack plugin to get valuable tips on how to fix some of the issues that occur during migration.

Note that this guide may change over time as we fix issues and find new ones. Please help us keep it complete – report any issues you encounter when switching to the IR compiler by submitting them to our issue tracker YouTrack or filling out this form.

Convert JS- and React-related classes and interfaces to external interfaces

Issue: Using Kotlin interfaces and classes (including data classes) that derive from pure JS classes, such as React's State and Props, can cause a ClassCastException. Such exceptions appear because the compiler attempts to work with instances of these classes as if they were Kotlin objects, when they actually come from JS.

Solution: convert all classes and interfaces that derive from pure JS classes to external interfaces:

```kotlin
// Replace this
interface AppState : State { }
interface AppProps : Props { }
data class CustomComponentState(var name: String) : State
```

```kotlin
// With this
external interface AppState : State { }
external interface AppProps : Props { }
external interface CustomComponentState : State {
   var name: String
}
```

In IntelliJ IDEA, you can use these structural search and replace templates to automatically mark interfaces as external:

- Template for State
- Template for Props

Convert properties of external interfaces to var

Issue: properties of external interfaces in Kotlin/JS code can't be read-only (val) properties because their values can be assigned only after the object is created with js() or jso() (a helper function from kotlin-wrappers):

```kotlin
import kotlinx.js.jso

val myState = jso<CustomComponentState>()
myState.name = "name"
```

Solution: convert all properties of external interfaces to var:

```kotlin
// Replace this
external interface CustomComponentState : State {
   val name: String
}
```

```kotlin
// With this
external interface CustomComponentState : State {
   var name: String
}
```

Convert functions with receivers in external interfaces to regular functions

Issue: external declarations can't contain functions with receivers, such as extension functions or properties with corresponding functional types.

Solution: convert such functions and properties to regular functions by adding the receiver object as an argument:

```kotlin
// Replace this
external interface ButtonProps : Props {
   var inside: StyledDOMBuilder<BUTTON>.() -> Unit
}
```

```kotlin
external interface ButtonProps : Props {
   var inside: (StyledDOMBuilder<BUTTON>) -> Unit
}
```

Create plain JS objects for interoperability

Issue: properties of a Kotlin object that implements an external interface are not enumerable. This means that they are not visible for operations that iterate over the object's properties, for example:

- for (var name in obj)

- console.log(obj)
- JSON.stringify(obj)

Although they are still accessible by the name: obj.myProperty

```
external interface AppProps { var name: String }
data class AppPropsImpl(override var name: String) : AppProps
fun main() {
   val jsApp = js("{name: 'App1'}") as AppProps // plain JS object
   println("Kotlin sees: ${jsApp.name}") // "App1"
   println("JSON.stringify sees:" + JSON.stringify(jsApp)) // {"name":"App1"} - OK

   val ktApp = AppPropsImpl("App2") // Kotlin object
   println("Kotlin sees: ${ktApp.name}") // "App2"
   // JSON sees only the backing field, not the property
   println("JSON.stringify sees:" + JSON.stringify(ktApp)) // {"_name_3":"App2"}
}
```

Solution 1: create plain JavaScript objects with js() or jso() (a helper function from kotlin-wrappers):

```
external interface AppProps { var name: String }
data class AppPropsImpl(override var name: String) : AppProps
```

```
// Replace this
val ktApp = AppPropsImpl("App1") // Kotlin object
```

```
// With this
val jsApp = js("{name: 'App1'}") as AppProps // or jso {}
```

Solution 2: create objects with kotlin.js.json():

```
// or with this
val jsonApp = kotlin.js.json(Pair("name", "App1")) as AppProps
```

Replace toString() calls on function references with .name

Issue: in the IR backend, calling toString() on function references doesn't produce unique values.

Solution: use the name property instead of toString().

Explicitly specify binaries.executable() in the build script

Issue: the compiler doesn't produce executable .js files.

This may happen because the default compiler produces JavaScript executables by default while the IR compiler needs an explicit instruction to do this. Learn more in the Kotlin/JS project setup instruction.

Solution: add the line binaries.executable() to the project's build.gradle(.kts).

```
kotlin {
    js(IR) {
        browser {
        }
        binaries.executable()
    }
}
```

Additional troubleshooting tips when working with the Kotlin/JS IR compiler

These hints may help you when troubleshooting problems in your projects using the Kotlin/JS IR compiler.

Make boolean properties nullable in external interfaces

Issue: when you call toString on a Boolean from an external interface, you're getting an error like Uncaught TypeError: Cannot read properties of undefined (reading 'toString'). JavaScript treats the null or undefined values of a boolean variable as false. If you rely on calling toString on a Boolean that may be null or undefined (for example when your code is called from JavaScript code you have no control over), be aware of this:

```
external interface SomeExternal {
    var visible: Boolean
}

fun main() {
    val empty: SomeExternal = js("{}")
    println(empty.visible.toString()) // Uncaught TypeError: Cannot read properties of undefined (reading 'toString')
}
```

Solution: you can make your Boolean properties of external interfaces nullable (Boolean?):

```
// Replace this
external interface SomeExternal {
    var visible: Boolean
}
```

```
// With this
external interface SomeExternal {
    var visible: Boolean?
}
```

Browser and DOM API

The Kotlin/JS standard library lets you access browser-specific functionality using the kotlinx.browser package, which includes typical top-level objects such as document and window. The standard library provides typesafe wrappers for the functionality exposed by these objects wherever possible. As a fallback, the dynamic type is used to provide interaction with functions that do not map well into the Kotlin type system.

Interaction with the DOM

For interaction with the Document Object Model (DOM), you can use the variable document. For example, you can set the background color of our website through this object:

```
document.bgColor = "FFAA12"
```

The document object also provides you a way to retrieve a specific element by ID, name, class name, tag name and so on. All returned elements are of type Element?. To access their properties, you need to cast them to their appropriate type. For example, assume that you have an HTML page with an email <input> field:

```
<body>
    <input type="text" name="email" id="email"/>

    <script type="text/javascript" src="tutorial.js"></script>
</body>
```

Note that your script is included at the bottom of the body tag. This ensures that the DOM is fully available before the script is loaded.

With this setup, you can access elements of the DOM. To access the properties of the input field, invoke getElementById and cast it to HTMLInputElement. You can then safely access its properties, such as value:

```
val email = document.getElementById("email") as HTMLInputElement
email.value = "hadi@jetbrains.com"
```

Much like you reference this input element, you can access other elements on the page, casting them to the appropriate types.

To see how to create and structure elements in the DOM in a concise way, check out the Typesafe HTML DSL.

Use JavaScript code from Kotlin

Kotlin was first designed for easy interoperation with the Java platform: it sees Java classes as Kotlin classes, and Java sees Kotlin classes as Java classes.

However, JavaScript is a dynamically typed language, which means it does not check types at compile time. You can freely talk to JavaScript from Kotlin via dynamic types. If you want to use the full power of the Kotlin type system, you can create external declarations for JavaScript libraries which will be understood by the Kotlin compiler and the surrounding tooling.

Inline JavaScript

You can inline some JavaScript code into your Kotlin code using the js() function. For example:

```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
```

Because the parameter of js is parsed at compile time and translated to JavaScript code "as-is", it is required to be a string constant. So, the following code is incorrect:

```
fun jsTypeOf(o: Any): String {
    return js(getTypeof() + " o") // error reported here
}
fun getTypeof() = "typeof"
```

Note that invoking js() returns a result of type dynamic, which provides no type safety at the compile time.

external modifier

To tell Kotlin that a certain declaration is written in pure JavaScript, you should mark it with the external modifier. When the compiler sees such a declaration, it assumes that the implementation for the corresponding class, function or property is provided externally (by the developer or via an npm dependency), and therefore does not try to generate any JavaScript code from the declaration. This is also why external declarations can't have a body. For example:

```
external fun alert(message: Any?): Unit

external class Node {
    val firstChild: Node

    fun append(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}

external val window: Window
```

Note that the external modifier is inherited by nested declarations. This is why in the example Node class, there is no external modifier before member functions and properties.

The external modifier is only allowed on package-level declarations. You can't declare an external member of a non-external class.

Declare (static) members of a class

In JavaScript you can define members either on a prototype or a class itself:

```
function MyClass() { ... }
MyClass.sharedMember = function() { /* implementation */ };
MyClass.prototype.ownMember = function() { /* implementation */ };
```

There is no such syntax in Kotlin. However, in Kotlin we have companion objects. Kotlin treats companion objects of external classes in a special way: instead of expecting an object, it assumes members of companion objects to be members of the class itself. MyClass from the example above can be described as follows:

```
external class MyClass {
```

```
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

Declare optional parameters

If you are writing an external declaration for a JavaScript function which has an optional parameter, use definedExternally. This delegates the generation of the default values to the JavaScript function itself:

```
external fun myFunWithOptionalArgs(
    x: Int,
    y: String = definedExternally,
    z: String = definedExternally
)
```

With this external declaration, you can call myFunWithOptionalArgs with one required argument and two optional arguments, where the default values are calculated by the JavaScript implementation of myFunWithOptionalArgs.

Extend JavaScript classes

You can easily extend JavaScript classes as if they were Kotlin classes. Just define an external open class and extend it by a non-external class. For example:

```
open external class Foo {
    open fun run()
    fun stop()
}

class Bar: Foo() {
    override fun run() {
        window.alert("Running!")
    }

    fun restart() {
        window.alert("Restarting")
    }
}
```

There are some limitations:

- When a function of an external base class is overloaded by signature, you can't override it in a derived class.
- You can't override a function with default arguments.
- Non-external classes can't be extended by external classes.

external interfaces

JavaScript does not have the concept of interfaces. When a function expects its parameter to support two methods foo and bar, you would just pass in an object that actually has these methods.

You can use interfaces to express this concept in statically typed Kotlin:

```
external interface HasFooAndBar {
    fun foo()

    fun bar()
}

external fun myFunction(p: HasFooAndBar)
```

A typical use case for external interfaces is to describe settings objects. For example:

```
external interface JQueryAjaxSettings {
    var async: Boolean

    var cache: Boolean
```

```
    var complete: (JQueryXHR, String) -> Unit

    // etc
}

fun JQueryAjaxSettings(): JQueryAjaxSettings = js("{}")

external class JQuery {
    companion object {
        fun get(settings: JQueryAjaxSettings): JQueryXHR
    }
}

fun sendQuery() {
    JQuery.get(JQueryAjaxSettings().apply {
        complete = { (xhr, data) ->
            window.alert("Request complete")
        }
    })
}
```

External interfaces have some restrictions:

- They can't be used on the right-hand side of is checks.
- They can't be passed as reified type arguments.
- They can't be used in class literal expressions (such as I::class).
- as casts to external interfaces always succeed. Casting to external interfaces produces the "Unchecked cast to external interface" compile time warning. The warning can be suppressed with the @Suppress("UNCHECKED_CAST_TO_EXTERNAL_INTERFACE") annotation.

  IntelliJ IDEA can also automatically generate the @Suppress annotation. Open the intentions menu via the light bulb icon or Alt-Enter, and click the small arrow next to the "Unchecked cast to external interface" inspection. Here, you can select the suppression scope, and your IDE will add the annotation to your file accordingly.

Casts

In addition to the "unsafe" cast operator as, which throws a ClassCastException in case a cast is not possible, Kotlin/JS also provides unsafeCast<T>(). When using unsafeCast, no type checking is done at all during runtime. For example, consider the following two methods:

```
fun usingUnsafeCast(s: Any) = s.unsafeCast<String>()
fun usingAsOperator(s: Any) = s as String
```

They will be compiled accordingly:

```
function usingUnsafeCast(s) {
    return s;
}

function usingAsOperator(s) {
    var tmp$;
    return typeof (tmp$ = s) === 'string' ? tmp$ : throwCCE();
}
```

Dynamic type

The dynamic type is not supported in code targeting the JVM.

Being a statically typed language, Kotlin still has to interoperate with untyped or loosely typed environments, such as the JavaScript ecosystem. To facilitate these use cases, the dynamic type is available in the language:

```
val dyn: dynamic = ...
```

The dynamic type basically turns off Kotlin's type checker:

- A value of the dynamic type can be assigned to any variable or passed anywhere as a parameter.
- Any value can be assigned to a variable of the dynamic type or passed to a function that takes dynamic as a parameter.
- null-checks are disabled for the dynamic type values.

The most peculiar feature of dynamic is that we are allowed to call any property or function with any parameters on a dynamic variable:

```
dyn.whatever(1, "foo", dyn) // 'whatever' is not defined anywhere
dyn.whatever(*arrayOf(1, 2, 3))
```

On the JavaScript platform this code will be compiled "as is": dyn.whatever(1) in Kotlin becomes dyn.whatever(1) in the generated JavaScript code.

When calling functions written in Kotlin on values of dynamic type, keep in mind the name mangling performed by the Kotlin to JavaScript compiler. You may need to use the @JsName annotation to assign well-defined names to the functions that you need to call.

A dynamic call always returns dynamic as a result, so you can chain such calls freely:

```
dyn.foo().bar.baz()
```

When you pass a lambda to a dynamic call, all of its parameters by default have the type dynamic:

```
dyn.foo {
    x -> x.bar() // x is dynamic
}
```

Expressions using values of dynamic type are translated to JavaScript "as is", and do not use the Kotlin operator conventions. The following operators are supported:

- binary: +, -, *, /, %, >, < >=, <=, ==, !=, ===, !==, &&, ||
- unary
  - prefix: -, +, !
  - prefix and postfix: ++, --
- assignments: +=, -=, *=, /=, %=
- indexed access:
  - read: d[a], more than one argument is an error
  - write: d[a1] = a2, more than one argument in [] is an error

in, !in and .. operations with values of type dynamic are forbidden.

For a more technical description, see the spec document.

Use dependencies from npm

In Kotlin/JS projects, all dependencies can be managed through the Gradle plugin. This includes Kotlin/Multiplatform libraries such as kotlinx.coroutines, kotlinx.serialization, or ktor-client.

For depending on JavaScript packages from npm, the Gradle DSL exposes an npm function that lets you specify packages you want to import from npm. Let's consider the import of an NPM package called is-sorted.

The corresponding part in the Gradle build file looks as follows:

```
dependencies {
    // ...
    implementation(npm("is-sorted", "1.0.5"))
}
```

Because JavaScript modules are usually dynamically typed and Kotlin is a statically typed language, you need to provide a kind of adapter. In Kotlin, such adapters

are called external declarations. For the is-sorted package which offers only one function, this declaration is small to write. Inside the source folder, create a new file called is-sorted.kt, and fill it with these contents:

```
@JsModule("is-sorted")
@JsNonModule
external fun <T> sorted(a: Array<T>): Boolean
```

Please note that if you're using CommonJS as a target, the @JsModule and @JsNonModule annotations need to be adjusted accordingly.

This JavaScript function can now be used just like a regular Kotlin function. Because we provided type information in the header file (as opposed to simply defining parameter and return type to be dynamic), proper compiler support and type-checking is also available.

```
console.log("Hello, Kotlin/JS!")
console.log(sorted(arrayOf(1,2,3)))
console.log(sorted(arrayOf(3,1,2)))
```

Running these three lines either in the browser or Node.js, the output shows that the call to sorted was properly mapped to the function exported by the is-sorted package:

```
Hello, Kotlin/JS!
true
false
```

Because the JavaScript ecosystem has multiple ways of exposing functions in a package (for example through named or default exports), other npm packages might need a slightly altered structure for their external declarations.

To learn more about how to write declarations, please refer to Calling JavaScript from Kotlin.

Use Kotlin code from JavaScript

Depending on the selected JavaScript Module system, the Kotlin/JS compiler generates different output. But in general, the Kotlin compiler generates normal JavaScript classes, functions and properties, which you can freely use from JavaScript code. There are some subtle things you should remember, though.

Isolating declarations in a separate JavaScript object in plain mode

If you have explicitly set your module kind to be plain, Kotlin creates an object that contains all Kotlin declarations from the current module. This is done to prevent spoiling the global object. This means that for a module myModule, all declarations are available to JavaScript via the myModule object. For example:

```
fun foo() = "Hello"
```

Can be called from JavaScript like this:

```
alert(myModule.foo());
```

This is not applicable when you compile your Kotlin module to JavaScript modules like UMD (which is the default setting for both browser and nodejs targets), CommonJS or AMD. In this case, your declarations will be exposed in the format specified by your chosen JavaScript module system. When using UMD or CommonJS, for example, your call site could look like this:

```
alert(require('myModule').foo());
```

Check the article on JavaScript Modules for more information on the topic of JavaScript module systems.

Package structure

Kotlin exposes its package structure to JavaScript, so unless you define your declarations in the root package, you have to use fully qualified names in JavaScript. For example:

```
package my.qualified.packagename

fun foo() = "Hello"
```

When using UMD or CommonJS, for example, your callsite could look like this:

```
alert(require('myModule').my.qualified.packagename.foo())
```

Or, in the case of using plain as a module system setting:

```
alert(myModule.my.qualified.packagename.foo());
```

@JsName annotation

In some cases (for example, to support overloads), the Kotlin compiler mangles the names of generated functions and attributes in JavaScript code. To control the generated names, you can use the @JsName annotation:

```
// Module 'kjs'
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
```

Now you can use this class from JavaScript in the following way:

```
// If necessary, import 'kjs' according to chosen module system
var person = new kjs.Person("Dmitry");   // refers to module 'kjs'
person.hello();                          // prints "Hello Dmitry!"
person.helloWithGreeting("Servus");      // prints "Servus Dmitry!"
```

If we didn't specify the @JsName annotation, the name of the corresponding function would contain a suffix calculated from the function signature, for example hello_61zpoe$.

Note that there are some cases in which the Kotlin compiler does not apply mangling:

- external declarations are not mangled.
- Any overridden functions in non-external classes inheriting from external classes are not mangled.

The parameter of @JsName is required to be a constant string literal which is a valid identifier. The compiler will report an error on any attempt to pass non-identifier string to @JsName. The following example produces a compile-time error:

```
@JsName("new C()")   // error here
external fun newC()
```

@JsExport annotation

> The @JsExport annotation is currently marked as experimental. Its design may change in future versions.

By applying the @JsExport annotation to a top-level declaration (like a class or function), you make the Kotlin declaration available from JavaScript. The annotation exports all nested declarations with the name given in Kotlin. It can also be applied on file-level using @file:JsExport.

To resolve ambiguities in exports (like overloads for functions with the same name), you can use the @JsExport annotation together with @JsName to specify the names for the generated and exported functions.

The @JsExport annotation is available in the current default compiler backend and the new IR compiler backend. If you are targeting the IR compiler backend, you must use the @JsExport annotation to make your functions visible from Kotlin in the first place.

For multiplatform projects, @JsExport is available in common code as well. It only has an effect when compiling for the JavaScript target, and allows you to also

export Kotlin declarations that are not platform specific.

Kotlin types in JavaScript

- Kotlin numeric types, except for kotlin.Long are mapped to JavaScript Number.
- kotlin.Char is mapped to JavaScript Number representing character code.
- Kotlin can't distinguish between numeric types at run time (except for kotlin.Long), so the following code works:

```
fun f() {
    val x: Int = 23
    val y: Any = x
    println(y as Float)
}
```

- Kotlin preserves overflow semantics for kotlin.Int, kotlin.Byte, kotlin.Short, kotlin.Char and kotlin.Long.
- kotlin.Long is not mapped to any JavaScript object, as there is no 64-bit integer number type in JavaScript. It is emulated by a Kotlin class.
- kotlin.String is mapped to JavaScript String.
- kotlin.Any is mapped to JavaScript Object (new Object(), {}, and so on).
- kotlin.Array is mapped to JavaScript Array.
- Kotlin collections (List, Set, Map, and so on) are not mapped to any specific JavaScript type.
- kotlin.Throwable is mapped to JavaScript Error.
- Kotlin preserves lazy object initialization in JavaScript.
- Kotlin does not implement lazy initialization of top-level properties in JavaScript.

Primitive arrays
Primitive array translation utilizes JavaScript TypedArray:

- kotlin.ByteArray, -.ShortArray, -.IntArray, -.FloatArray, and -.DoubleArray are mapped to JavaScript Int8Array, Int16Array, Int32Array, Float32Array, and Float64Array correspondingly.
- kotlin.BooleanArray is mapped to JavaScript Int8Array with a property $type$ == "BooleanArray".
- kotlin.CharArray is mapped to JavaScript UInt16Array with a property $type$ == "CharArray".
- kotlin.LongArray is mapped to JavaScript Array of kotlin.Long with a property $type$ == "LongArray".

JavaScript modules

You can compile your Kotlin projects to JavaScript modules for various popular module systems. We currently support the following configurations for JavaScript modules:

- Unified Module Definitions (UMD), which is compatible with both AMD and CommonJS. UMD modules are also able to be executed without being imported or when no module system is present. This is the default option for the browser and nodejs targets.
- Asynchronous Module Definitions (AMD), which is in particular used by the RequireJS library.
- CommonJS, widely used by Node.js/npm (require function and module.exports object)
- Plain. Don't compile for any module system. You can access a module by its name in the global scope.

Browser targets

If you're targeting the browser and want to use a different module system than UMD, you can specify the desired module type in the webpackTask configuration block. For example, to switch to CommonJS, use:

```
kotlin {
    js {
        browser {
            webpackTask {
                output.libraryTarget = "commonjs2"
            }
        }
        binaries.executable()
    }
}
```

Webpack provides two different "flavors" of CommonJS, commonjs and commonjs2, which affect the way your declarations are made available. While in most cases, you probably want commonjs2, which adds the module.exports syntax to the generated library, you can also opt for the "pure" commonjs option, which implements the CommonJS specification exactly. To learn more about the difference between commonjs and commonjs2, check here.

JavaScript libraries and Node.js files

If you are creating a library that will be consumed from JavaScript or a Node.js file, and want to use a different module system, the instructions are slightly different.

Choose the target module system

To select module kind, set the moduleKind compiler option in the Gradle build script.

Kotlin

```
tasks.named<KotlinJsCompile>("compileKotlinJs").configure {
    compilerOptions.moduleKind.set(org.jetbrains.kotlin.gradle.dsl.JsModuleKind.MODULE_COMMONJS)
}
```

Groovy

```
compileKotlinJs.compilerOptions.moduleKind = org.jetbrains.kotlin.gradle.dsl.JsModuleKind.MODULE_COMMONJS
```

Available values are: umd (default), commonjs, amd, plain.

> This is different from adjusting webpackTask.output.libraryTarget. The library target changes the output generated by webpack (after your code has already been compiled). compilerOptions.moduleKind changes the output generated by the Kotlin compiler.

In the Kotlin Gradle DSL, there is also a shortcut for setting the CommonJS module kind:

```
kotlin {
    js {
        useCommonJs()
        // ...
    }
}
```

@JsModule annotation

To tell Kotlin that an external class, package, function or property is a JavaScript module, you can use @JsModule annotation. Consider you have the following CommonJS module called "hello":

```
module.exports.sayHello = function(name) { alert("Hello, " + name); }
```

You should declare it like this in Kotlin:

```
@JsModule("hello")
external fun sayHello(name: String)
```

Apply @JsModule to packages

Some JavaScript libraries export packages (namespaces) instead of functions and classes. In terms of JavaScript, it's an object that has members that are classes, functions and properties. Importing these packages as Kotlin objects often looks unnatural. The compiler can map imported JavaScript packages to Kotlin packages, using the following notation:

```
@file:JsModule("extModule")
package ext.jspackage.name

external fun foo()

external class C
```

where the corresponding JavaScript module is declared like this:

```
module.exports = {
    foo:  { /* some code here */ },
    C:  { /* some code here */ }
}
```

Files marked with @file:JsModule annotation can't declare non-external members. The example below produces a compile-time error:

```
@file:JsModule("extModule")
package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // error here
```

Import deeper package hierarchies

In the previous example the JavaScript module exports a single package. However, some JavaScript libraries export multiple packages from within a module. This case is also supported by Kotlin, though you have to declare a new .kt file for each package you import.

For example, let's make the example a bit more complicated:

```
module.exports = {
    mylib: {
        pkg1: {
            foo: function() { /* some code here */ },
            bar: function() { /* some code here */ }
        },
        pkg2: {
            baz: function() { /* some code here */ }
        }
    }
}
```

To import this module in Kotlin, you have to write two Kotlin source files:

```
@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg1")
package extlib.pkg1

external fun foo()

external fun bar()
```

and

```
@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg2")
package extlib.pkg2
```

```
external fun baz()
```

@JsNonModule annotation

When a declaration is marked as @JsModule, you can't use it from Kotlin code when you don't compile it to a JavaScript module. Usually, developers distribute their libraries both as JavaScript modules and downloadable .js files that you can copy to your project's static resources and include via a <script> tag. To tell Kotlin that it's okay to use a @JsModule declaration from a non-module environment, add the @JsNonModule annotation. For example, consider the following JavaScript code:

```
function topLevelSayHello(name) { alert("Hello, " + name); }
if (module && module.exports) {
    module.exports = topLevelSayHello;
}
```

You could describe it from Kotlin as follows:

```
@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)
```

Module system used by the Kotlin Standard Library

Kotlin is distributed with the Kotlin/JS standard library as a single file, which is itself compiled as an UMD module, so you can use it with any module system described above. While for most use cases of Kotlin/JS, it is recommended to use a Gradle dependency on kotlin-stdlib-js, it is also available on NPM as the kotlin package.

Kotlin/JS reflection

Kotlin/JS provides a limited support for the Kotlin reflection API. The only supported parts of the API are:

- class references (::class).
- KType and typeof() function.

Class references

The ::class syntax returns a reference to the class of an instance, or the class corresponding to the given type. In Kotlin/JS, the value of a ::class expression is a stripped-down KClass implementation that supports only:

- simpleName and isInstance() members.
- cast() and safeCast() extension functions.

In addition to that, you can use KClass.js to access the JsClass instance corresponding to the class. The JsClass instance itself is a reference to the constructor function. This can be used to interoperate with JS functions that expect a reference to a constructor.

KType and typeOf()

The typeof() function constructs an instance of KType for a given type. The KType API is fully supported in Kotlin/JS except for Java-specific parts.

Example

Here is an example of the reflection usage in Kotlin/JS.

```
open class Shape
class Rectangle : Shape()

inline fun <reified T> accessReifiedTypeArg() =
```

```
    println(typeOf<T>().toString())

fun main() {
    val s = Shape()
    val r = Rectangle()

    println(r::class.simpleName) // Prints "Rectangle"
    println(Shape::class.simpleName) // Prints "Shape"
    println(Shape::class.js.name) // Prints "Shape"

    println(Shape::class.isInstance(r)) // Prints "true"
    println(Rectangle::class.isInstance(s)) // Prints "false"
    val rShape = Shape::class.cast(r) // Casts a Rectangle "r" to Shape

    accessReifiedTypeArg<Rectangle>() // Accesses the type via typeOf(). Prints "Rectangle"
}
```

Typesafe HTML DSL

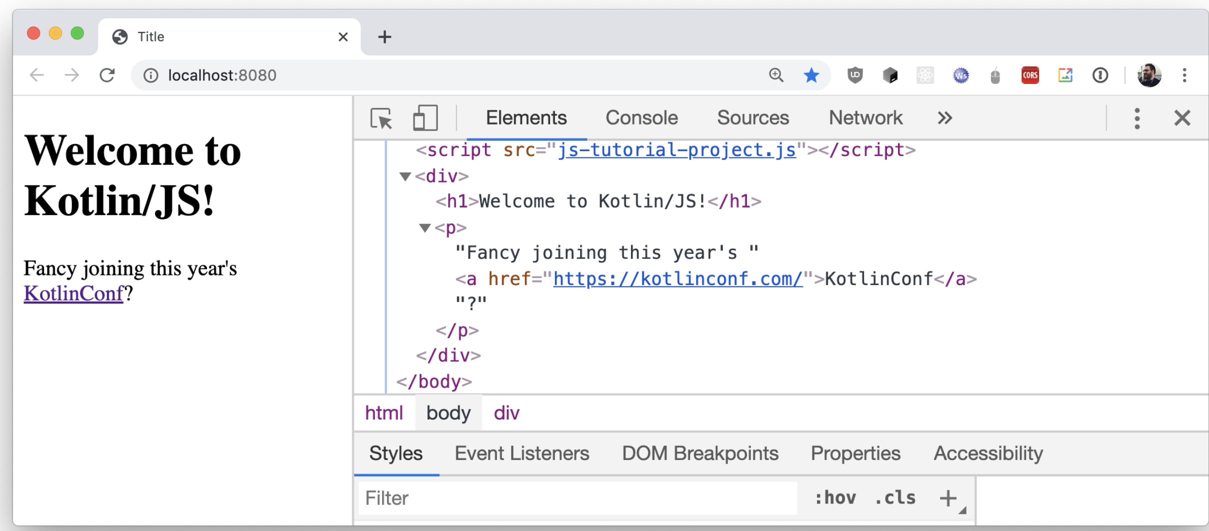
The kotlinx.html library provides the ability to generate DOM elements using statically typed HTML builders (and besides JavaScript, it is even available on the JVM target!) To use the library, include the corresponding repository and dependency to our build.gradle.kts file:

```
repositories {
    // ...
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-js"))
    implementation("org.jetbrains.kotlinx:kotlinx-html-js:0.8.0")
    // ...
}
```

Once the dependency is included, you can access the different interfaces provided to generate the DOM. To render a headline, some text, and a link, the following snippet would be sufficient, for example:

```
import kotlinx.browser.*
import kotlinx.html.*
import kotlinx.html.dom.*

fun main() {
    document.body!!.append.div {
        h1 {
            +"Welcome to Kotlin/JS!"
        }
        p {
            +"Fancy joining this year's "
            a("https://kotlinconf.com/") {
                +"KotlinConf"
            }
            +"?"
        }
    }
}
```

When running this example in the browser, the DOM will be assembled in a straightforward way. This is easily confirmed by checking the Elements of the website using the developer tools of our browser:



Rendering a website from kotlinx.html

To learn more about the kotlinx.html library, check out the GitHub Wiki, where you can find more information about how to create elements without adding them to the DOM, binding to events like onClick, and examples on how to apply CSS classes to your HTML elements, to name just a few.

Build a web application with React and Kotlin/JS — tutorial

This tutorial will teach you how to build a browser application with Kotlin/JS and the React framework. You will:

- Complete common tasks associated with building a typical React application.
- Explore how Kotlin's DSLs can be used to help express concepts concisely and uniformly without sacrificing readability, allowing you to write a full-fledged application completely in Kotlin.
- Learn how to use ready-made npm components, use external libraries, and publish the final application.

The output will be a KotlinConf Explorer web app dedicated to the KotlinConf event, with links to conference talks. Users will be able to watch all the talks on one page and mark them as seen or unseen.

The tutorial assumes you have prior knowledge of Kotlin and basic knowledge of HTML and CSS. Understanding the basic concepts behind React may help you understand some sample code, but it is not strictly required.

> You can get the final application here.

Before you start

1. Download and install the latest version of IntelliJ IDEA.
2. Clone the project template and open it in IntelliJ IDEA. The template includes a basic Kotlin Multiplatform Gradle project with all required configurations and dependencies

   - Dependencies and tasks in the build.gradle.kts file:

```
dependencies {
    // React, React DOM + Wrappers
    implementation(enforcedPlatform("org.jetbrains.kotlin-wrappers:kotlin-wrappers-bom:1.0.0-pre.430"))
    implementation("org.jetbrains.kotlin-wrappers:kotlin-react")
    implementation("org.jetbrains.kotlin-wrappers:kotlin-react-dom")

    // Kotlin React Emotion (CSS)
```

```
    implementation("org.jetbrains.kotlin-wrappers:kotlin-emotion")

    // Video Player
    implementation(npm("react-player", "2.12.0"))

    // Share Buttons
    implementation(npm("react-share", "4.4.1"))

    // Coroutines & serialization
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4")
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.0")
}
```

- An HTML template page in src/jsMain/resources/index.html for inserting JavaScript code that you'll be using in this tutorial:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello, Kotlin/JS!</title>
</head>
<body>
    <div id="root"></div>
    <script src="confexplorer.js"></script>
</body>
</html>
```

  Kotlin/JS projects are automatically bundled with all of your code and its dependencies into a single JavaScript file with the same name as the project, confexplorer.js, when you build them. As a typical JavaScript convention, the content of the body (including the root div) is loaded first to ensure that the browser loads all page elements before the scripts.

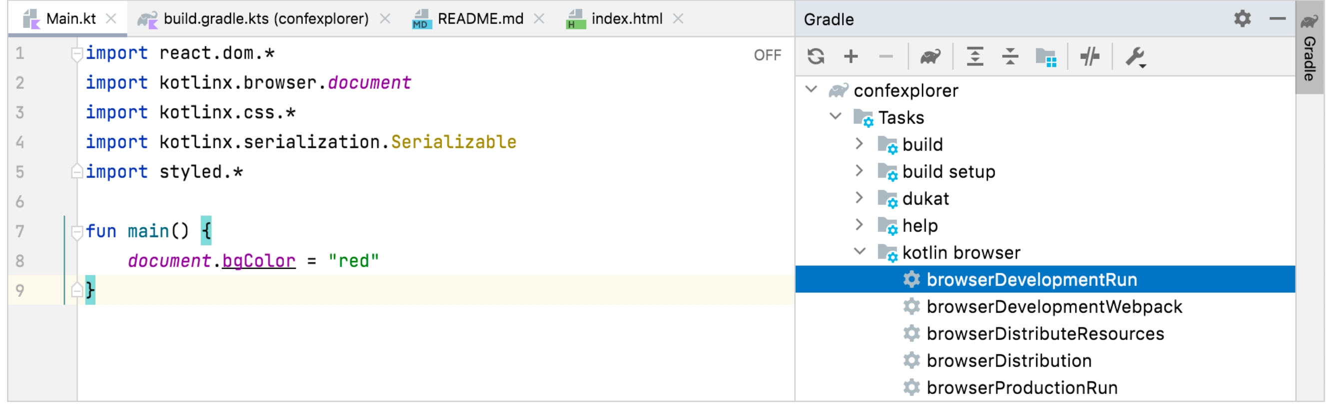
- A code snippet in src/jsMain/kotlin/Main.kt:

```
import kotlinx.browser.document

fun main() {
    document.bgColor = "red"
}
```

Run the development server

By default, the Kotlin Multiplatform Gradle plugin comes with support for an embedded webpack-dev-server, allowing you to run the application from the IDE without manually setting up any servers.

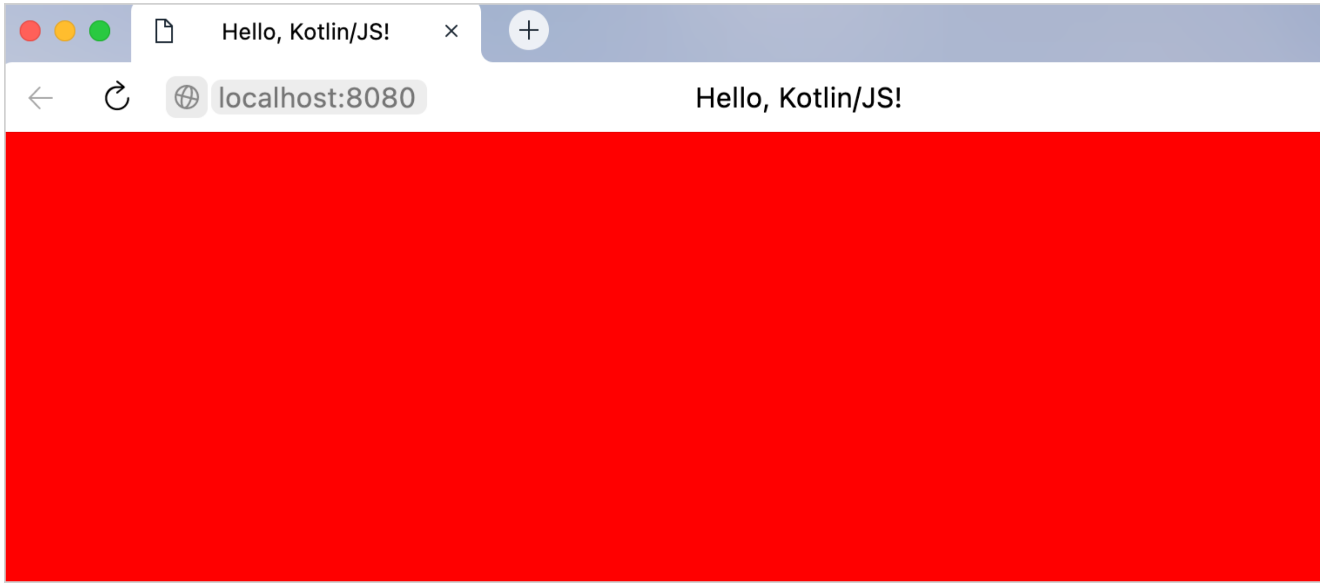
To test that the program successfully runs in the browser, start the development server by invoking the run or browserDevelopmentRun task (available in the other or kotlin browser directory) from the Gradle tool window inside IntelliJ IDEA:



Gradle tasks list

To run the program from the Terminal, use ./gradlew run instead.

When the project is compiled and bundled, a blank red page will appear in a browser window:

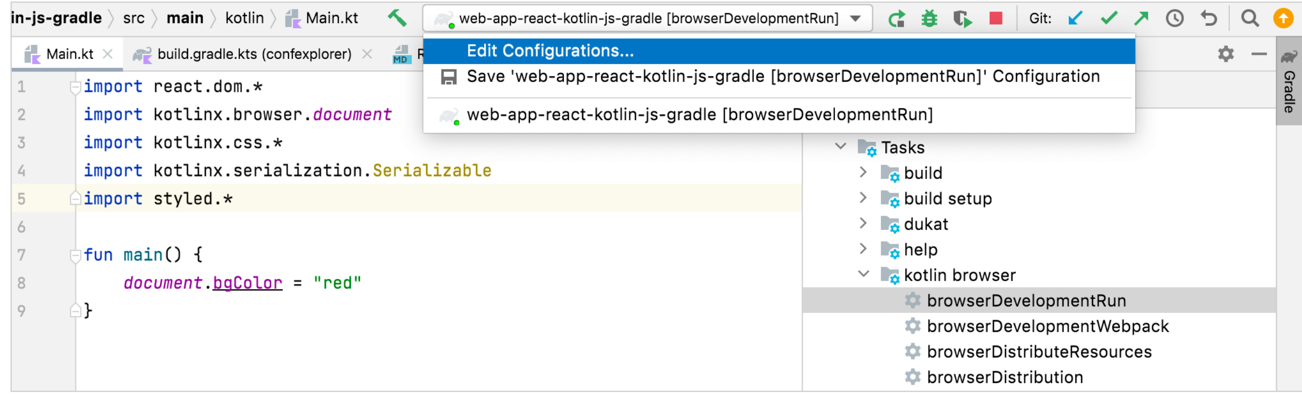


Blank red page

Enable hot reload / continuous mode

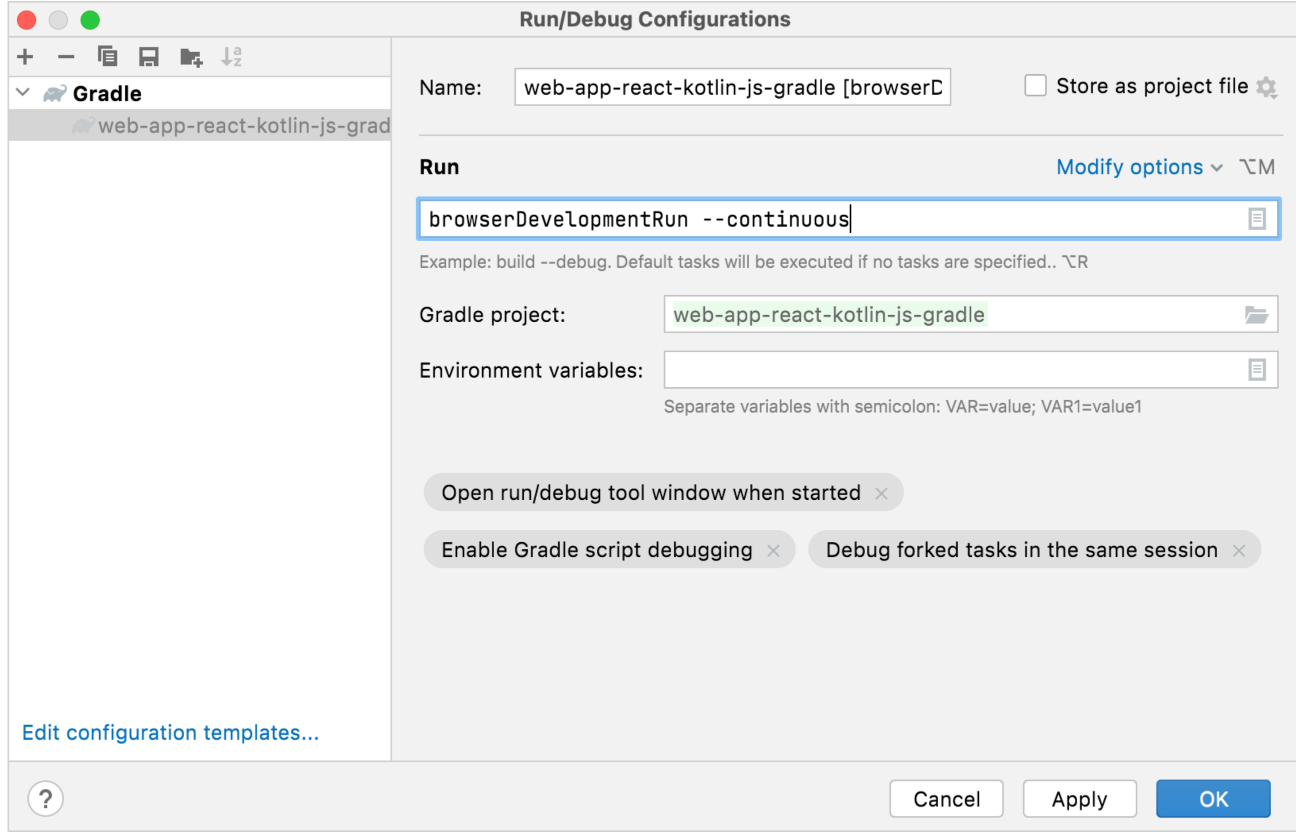
Configure continuous compilation mode so you don't have to manually compile and execute your project every time you make changes. Make sure to stop all running development server instances before proceeding.

1. Edit the run configuration that IntelliJ IDEA automatically generates after running the Gradle run task for the first time:



Edit a run configuration

2. In the Run/Debug Configurations dialog, add the --continuous option to the arguments for the run configuration:



Enable continuous mode

After applying the changes, you can use the Run button inside IntelliJ IDEA to start the development server back up. To run the continuous Gradle builds from the Terminal, use ./gradlew run --continuous instead.

3. To test this feature, change the color of the page to blue in the Main.kt file while the Gradle task is running:

```
document.bgColor = "blue"
```

The project then recompiles, and after a reload the browser page will be the new color.

You can keep the development server running in continuous mode during the development process. It will automatically rebuild and reload the page when you make changes.

> You can find this state of the project on the master branch here.

Create a web app draft

Add the first static page with React

To make your app display a simple message, replace the code in the Main.kt file with the following:

```
import kotlinx.browser.document
import react.*
import emotion.react.css
import csstype.Position
import csstype.px
import react.dom.html.ReactHTML.h1
import react.dom.html.ReactHTML.h3
import react.dom.html.ReactHTML.div
import react.dom.html.ReactHTML.p
import react.dom.html.ReactHTML.img
import react.dom.client.createRoot
```

```
import kotlinx.serialization.Serializable

fun main() {
    val container = document.getElementById("root") ?: error("Couldn't find root container!")
    createRoot(container).render(Fragment.create {
        h1 {
            +"Hello, React+Kotlin/JS!"
        }
    })
}
```

- The render() function instructs kotlin-react-dom to render the first HTML element inside a fragment to the root element. This element is a container defined in src/jsMain/resources/index.html, which was included in the template.
- The content is an <h1> header and uses a typesafe DSL to render HTML.
- h1 is a function that takes a lambda parameter. When you add the + sign in front of a string literal, the unaryPlus() function is actually invoked using operator overloading. It appends the string to the enclosed HTML element.

When the project recompiles, the browser displays this HTML page:

Hello, React+Kotlin/JS!

An HTML page example

Convert HTML to Kotlin's typesafe HTML DSL

The Kotlin wrappers for React come with a domain-specific language (DSL) that makes it possible to write HTML in pure Kotlin code. In this way, it's similar to JSX from JavaScript. However, with this markup being Kotlin, you get all the benefits of a statically typed language, such as autocomplete or type checking.

Compare the classic HTML code for your future web app and its typesafe variant in Kotlin:

HTML

```
<h1>KotlinConf Explorer</h1>
<div>
    <h3>Videos to watch</h3>
    <p>John Doe: Building and breaking things</p>
    <p>Jane Smith: The development process</p>
    <p>Matt Miller: The Web 7.0</p>
    <h3>Videos watched</h3>
    <p>Tom Jerry: Mouseless development</p>
</div>
<div>
    <h3>John Doe: Building and breaking things</h3>
    <img src="https://via.placeholder.com/640x360.png?text=Video+Player+Placeholder">
</div>
```

Kotlin

```
h1 {
    +"KotlinConf Explorer"
}
div {
    h3 {
        +"Videos to watch"
    }
    p {
        + "John Doe: Building and breaking things"
    }
    p {
        +"Jane Smith: The development process"
    }
    p {
```

```
            +"Matt Miller: The Web 7.0"
        }
        h3 {
            +"Videos watched"
        }
        p {
            +"Tom Jerry: Mouseless development"
        }
    }
    div {
        h3 {
            +"John Doe: Building and breaking things"
        }
        img {
           src = "https://via.placeholder.com/640x360.png?text=Video+Player+Placeholder"
        }
    }
```

Copy the Kotlin code and update the Fragment.create() function call inside the main() function, replacing the previous h1 tag.

Wait for the browser to reload. The page should now look like this:

KotlinConf Explorer

Videos to watch

John Doe: Building and breaking things

Jane Smith: The development process

Matt Miller: The Web 7.0

Videos watched

Tom Jerry: Mouseless development

John Doe: Building and breaking things

Video Player Placeholder

The web app draft

Add videos using Kotlin constructs in markup

There are some advantages to writing HTML in Kotlin using this DSL. You can manipulate your app using regular Kotlin constructs, like loops, conditions, collections, and string interpolation.

You can now replace the hardcoded list of videos with a list of Kotlin objects:

1. In Main.kt, create a Video data class to keep all video attributes in one place:

```
data class Video(
    val id: Int,
    val title: String,
```

```
    val speaker: String,
    val videoUrl: String
)
```

2. Fill up the two lists, for unwatched videos and watched videos, respectively. Add these declarations at file-level in Main.kt:

```
val unwatchedVideos = listOf(
    Video(1, "Opening Keynote", "Andrey Breslav", "https://youtu.be/PsaFVLr8t4E"),
    Video(2, "Dissecting the stdlib", "Huyen Tue Dao", "https://youtu.be/Fzt_9I733Yg"),
    Video(3, "Kotlin and Spring Boot", "Nicolas Frankel", "https://youtu.be/pSiZVAeReeg")
)

val watchedVideos = listOf(
    Video(4, "Creating Internal DSLs in Kotlin", "Venkat Subramaniam", "https://youtu.be/JzTeAM8N1-o")
)
```

3. To use these videos on the page, write a Kotlin for loop to iterate over the collection of unwatched Video objects. Replace the three p tags under "Videos to watch" with the following snippet:

```
for (video in unwatchedVideos) {
    p {
        +"${video.speaker}: ${video.title}"
    }
}
```

4. Apply the same process to modify the code for the single tag following "Videos watched" as well:

```
for (video in watchedVideos) {
    p {
        +"${video.speaker}: ${video.title}"
    }
}
```

Wait for the browser to reload. The layout should stay the same as before. You can add some more videos to the list to make sure that the loop is working.

Add styles with typesafe CSS

The kotlin-emotion wrapper for the Emotion library makes it possible to specify CSS attributes – even dynamic ones – right alongside HTML with JavaScript. Conceptually, that makes it similar to CSS-in-JS – but for Kotlin. The benefit of using a DSL is that you can use Kotlin code constructs to express formatting rules.

The template project for this tutorial already includes the dependency needed to use kotlin-emotion:

```
dependencies {
    // ...
    // Kotlin React Emotion (CSS) (chapter 3)
    implementation("org.jetbrains.kotlin-wrappers:kotlin-emotion")
    // ...
}
```

With kotlin-emotion, you can specify a css block inside HTML elements div and h3, where you can define the styles.

To move the video player to the top right-hand corner of the page, use CSS and adjust the code for the video player (the last div in the snippet):

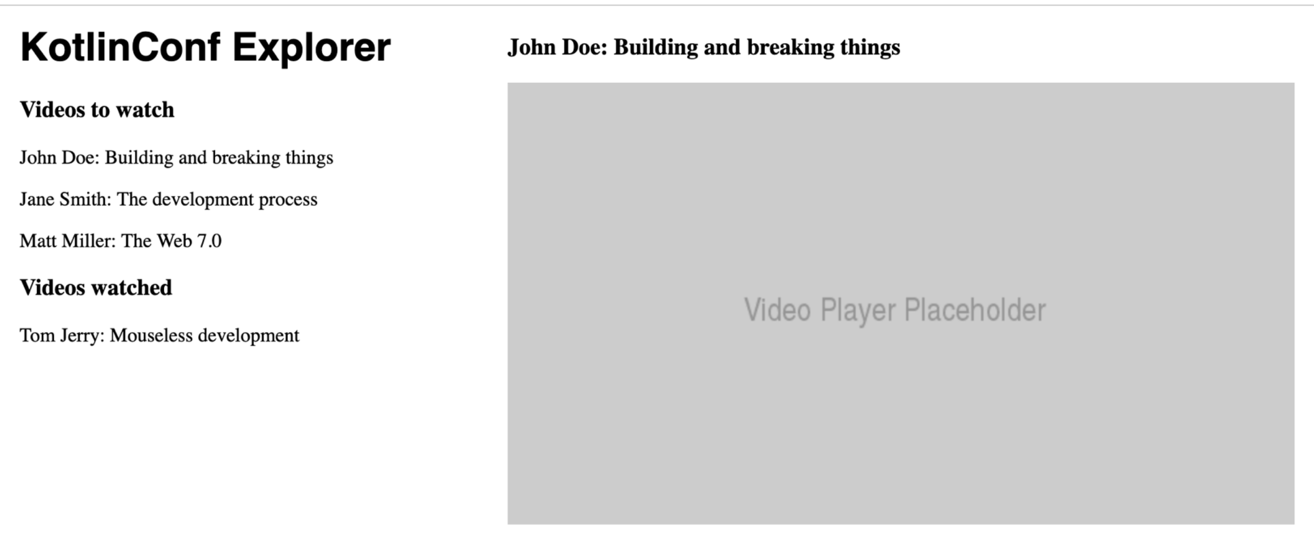
```
div {
    css {
        position = Position.absolute
        top = 10.px
        right = 10.px
    }
    h3 {
        +"John Doe: Building and breaking things"
    }
    img {
        src = "https://via.placeholder.com/640x360.png?text=Video+Player+Placeholder"
    }
}
```

Feel free to experiment with some other styles. For example, you could change the fontFamily or add some color to your UI.

Design app components

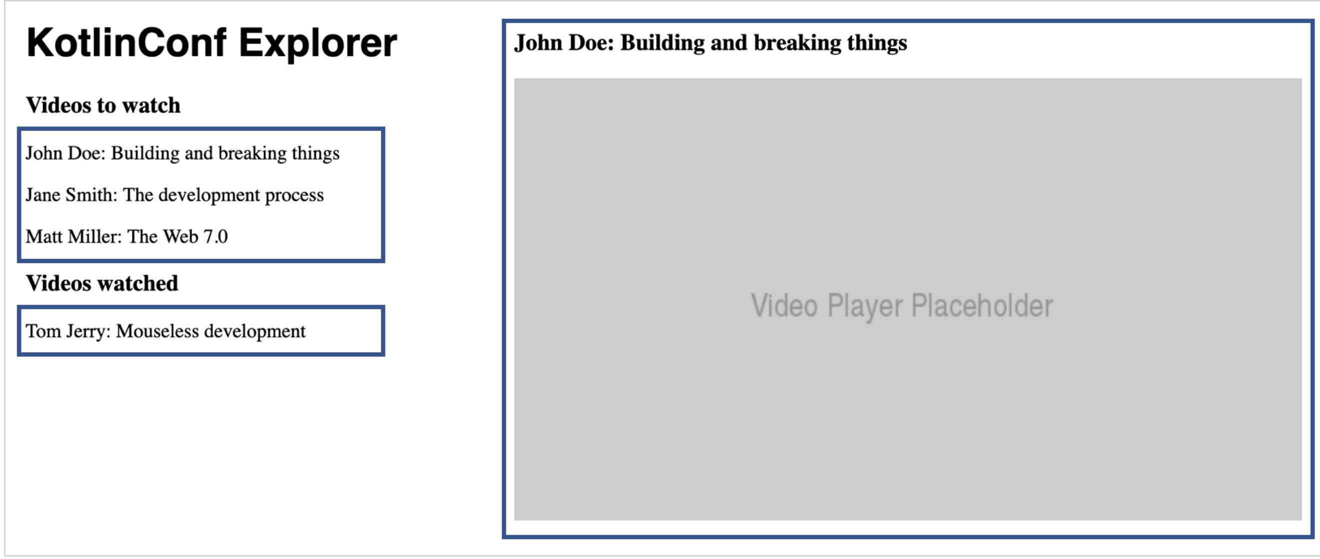
The basic building blocks in React are called components. Components themselves can also be composed of other, smaller components. By combining components, you build your application. If you structure components to be generic and reusable, you'll be able to use them in multiple parts of the app without duplicating code or logic.

The content of the render() function generally describes a basic component. The current layout of your application looks like this:



Current layout

If you decompose your application into individual components, you'll end up with a more structured layout in which each component handles its responsibilities:



Structured layout with components

Components encapsulate a particular functionality. Using components shortens source code and makes it easier to read and understand.

Add the main component

To start creating the application's structure, first explicitly specify App, the main component for rendering to the rootelement:

1. Create a new App.kt file in the src/jsMain/kotlin folder.
2. Inside this file, add the following snippet and move the typesafe HTML from Main.kt into it:

```kotlin
import kotlinx.coroutines.async
import react.*
import react.dom.*
import kotlinx.browser.window
import kotlinx.coroutines.*
import kotlinx.serialization.decodeFromString
import kotlinx.serialization.json.Json
import emotion.react.css
import csstype.Position
import csstype.px
import react.dom.html.ReactHTML.h1
import react.dom.html.ReactHTML.h3
import react.dom.html.ReactHTML.div
import react.dom.html.ReactHTML.p
import react.dom.html.ReactHTML.img

val App = FC<Props> {
    // typesafe HTML goes here, starting with the first h1 tag!
}
```

The FC function creates a function component.

3. In the Main.kt file, update the main() function as follows:

```kotlin
fun main() {
    val container = document.getElementById("root") ?: error("Couldn't find root container!")
    createRoot(container).render(App.create())
}
```

Now the program creates an instance of the App component and renders it to the specified container.

For more information about React concepts, see the documentation and guides.

Extract a list component

Since the watchedVideos and unwatchedVideos lists each contain a list of videos, it makes sense to create a single reusable component, and only adjust the content displayed in the lists.

The VideoList component follows the same pattern as the App component. It uses the FC builder function, and contains the code from the unwatchedVideos list.

1. Create a new VideoList.kt file in the src/jsMain/kotlin folder and add the following code:

```kotlin
import kotlinx.browser.window
import react.*
import react.dom.*
import react.dom.html.ReactHTML.p

val VideoList = FC<Props> {
    for (video in unwatchedVideos) {
        p {
            +"${video.speaker}: ${video.title}"
        }
    }
}
```

2. In App.kt, use the VideoList component by invoking it without parameters:

```kotlin
// . . .

div {
    h3 {
        +"Videos to watch"
    }
    VideoList()

    h3 {
        +"Videos watched"
    }
    VideoList()
}

// . . .
```

For now, the App component has no control over the content that is shown by the VideoList component. It's hard-coded, so you see the same list twice.

Add props to pass data between components

Since you're going to reuse the VideoList component, you'll need to be able to fill it with different content. You can add the ability to pass the list of items as an attribute to the component. In React, these attributes are called props. When the props of a component are changed in React, the framework automatically re-renders the component.

For VideoList, you'll need a prop containing the list of videos to be shown. Define an interface that holds all the props which can be passed to a VideoList component:

1. Add the following definition to the VideoList.kt file:

```
external interface VideoListProps : Props {
    var videos: List<Video>
}
```

   The external modifier tells the compiler that the interface's implementation is provided externally, so it doesn't try to generate JavaScript code from the declaration.

2. Adjust the class definition of VideoList to make use of the props that are passed into the FC block as a parameter:

```
val VideoList = FC<VideoListProps> { props ->
    for (video in props.videos) {
        p {
            key = video.id.toString()
            +"${video.speaker}: ${video.title}"
        }
    }
}
```

   The key attribute helps the React renderer figure out what to do when the value of props.videos changes. It uses the key to determine which parts of a list need to be refreshed and which ones stay the same. You can find more information about lists and keys in the React guide.

3. In the App component, make sure that the child components are instantiated with the proper attributes. In App.kt, replace the two loops underneath the h3 elements with an invocation of VideoList together with the attributes for unwatchedVideos and watchedVideos. In the Kotlin DSL, you assign them inside a block belonging to the VideoList component:

```
h3 {
    +"Videos to watch"
}
VideoList {
    videos = unwatchedVideos
}
h3 {
    +"Videos watched"
}
VideoList {
    videos = watchedVideos
}
```

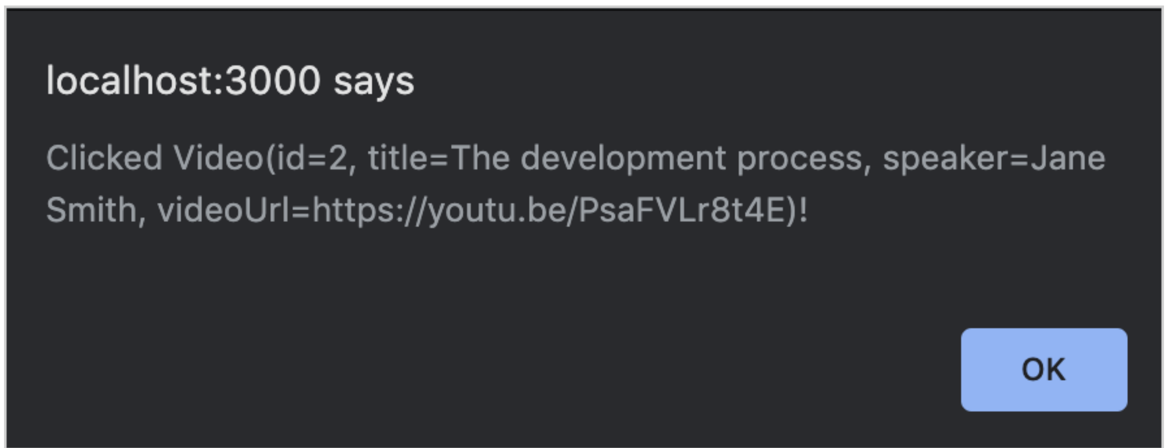
After a reload, the browser will show that the lists now render correctly.

Make the list interactive

First, add an alert message that pops up when users click on a list entry. In VideoList.kt, add an onClick handler function that triggers an alert with the current video:

```
// . . .

p {
    key = video.id.toString()
    onClick = {
        window.alert("Clicked $video!")
    }
    +"${video.speaker}: ${video.title}"
}

// . . .
```

If you click on one of the list items in the browser window, you'll get information about the video in an alert window like this:



Browser alert window

> Defining an onClick function directly as lambda is concise and very useful for prototyping. However, due to the way equality currently works in Kotlin/JS, performance-wise it's not the most optimized way to pass click handlers. If you want to optimize rendering performance, consider storing your functions in a variable and passing them.

Add state to keep values

Instead of just alerting the user, you can add some functionality for highlighting the selected video with a ▶ triangle. To do that, introduce some state specific to this component.

State is one of core concepts in React. In modern React (which uses the so-called Hooks API), state is expressed using the useState hook.

1. Add the following code to the top of the VideoList declaration:

```
val VideoList = FC<VideoListProps> { props ->
    var selectedVideo: Video? by useState(null)

// . . .
```

- The VideoList functional component keeps state (a value that is independent of the current function invocation). State is nullable, and has the Video? type. Its default value is null.
- The useState() function from React instructs the framework to keep track of state across multiple invocations of the function. For example, even though you specify a default value, React makes sure that the default value is only assigned in the beginning. When state changes, the component will re-render based on the new state.
- The by keyword indicates that useState() acts as a delegated property. Like with any other variable, you read and write values. The implementation behind useState() takes care of the machinery required to make state work.

To learn more about the State Hook, check out the React documentation.

2. Change your implementation of the VideoList component to look as follows:

```
val VideoList = FC<VideoListProps> { props ->
    var selectedVideo: Video? by useState(null)
    for (video in props.videos) {
        p {
            key = video.id.toString()
            onClick = {
                selectedVideo = video
            }
            if (video == selectedVideo) {
                +"▶ "
            }
            +"${video.speaker}: ${video.title}"
        }
```

```
    }
}
```

- When the user clicks a video, its value is assigned to the selectedVideo variable.
- When the selected list entry is rendered, the triangle is prepended.

You can find more details about state management in the React FAQ.

Check the browser and click an item in the list to make sure that everything is working correctly.

Compose components

Currently, the two video lists work on their own, meaning that each list keeps track of a selected video. Users can select two videos, one in the unwatched list and one in watched, even though there's only one player:

Two videos are selected in both lists simultaneously

A list can't keep track of which video is selected both inside itself, and inside a sibling list. The reason is that the selected video is part not of the list state, but of the application state. This means you need to lift state out of the individual components.

Lift state

React makes sure that props can only be passed from a parent component to its children. This prevents components from being hard-wired together.

If a component wants to change state of a sibling component, it needs to do so via its parent. At that point, state also no longer belongs to any of the child components but to the overarching parent component.

The process of migrating state from components to their parents is called lifting state. For your app, add currentVideo as state to the App component:

1. In App.kt, add the following to the top of the definition of the App component:

   ```
   val App = FC<Props> {
       var currentVideo: Video? by useState(null)

       // . . .
   }
   ```

   The VideoList component no longer needs to keep track of state. It will receive the current video as a prop instead.

2. Remove the useState() call in VideoList.kt.

3. Prepare the VideoList component to receive the selected video as a prop. To do so, expand the VideoListProps interface to contain the selectedVideo:

```kotlin
external interface VideoListProps : Props {
    var videos: List<Video>
    var selectedVideo: Video?
}
```

4. Change the condition of the triangle so that it uses props instead of state:

```kotlin
if (video == props.selectedVideo) {
    +"▶ "
}
```

Pass handlers

At the moment, there's no way to assign a value to a prop, so the onClick function won't work the way it is currently set up. To change state of a parent component, you need to lift state again.

In React, state always flows from parent to child. So, to change the application state from one of the child components, you need to move the logic for handling user interaction to the parent component and then pass the logic in as a prop. Remember that in Kotlin, variables can have the type of a function.

1. Expand the VideoListProps interface again so that it contains a variable onSelectVideo, which is a function that takes a Video and returns Unit:

```kotlin
external interface VideoListProps : Props {
    // ...
    var onSelectVideo: (Video) -> Unit
}
```

2. In the VideoList component, use the new prop in the onClick handler:

```kotlin
onClick = {
    props.onSelectVideo(video)
}
```

3. You can now go back to the App component and pass selectedVideo and a handler for onSelectVideo for each of the two video lists:

```kotlin
VideoList {
    videos = unwatchedVideos // and watchedVideos respectively
    selectedVideo = currentVideo
    onSelectVideo = { video ->
        currentVideo = video
    }
}
```

4. Repeat the previous step for the watched videos list.

Switch back to your browser and make sure that when selecting a video the selection jumps between the two lists without duplication.

Add more components

Extract the video player component

You can now create another self-contained component, a video player, which is currently a placeholder image. Your video player needs to know the talk title, the author of the talk, and the link to the video. This information is already contained in each Video object, so you can pass it as a prop and access its attributes.

1. Create a new VideoPlayer.kt file and add the following implementation for the VideoPlayer component:

```kotlin
import csstype.*
import react.*
import emotion.react.css
import react.dom.html.ReactHTML.button
import react.dom.html.ReactHTML.div
import react.dom.html.ReactHTML.h3
import react.dom.html.ReactHTML.img

external interface VideoPlayerProps : Props {
    var video: Video
```

```
}

val VideoPlayer = FC<VideoPlayerProps> { props ->
    div {
        css {
            position = Position.absolute
            top = 10.px
            right = 10.px
        }
        h3 {
            +"${props.video.speaker}: ${props.video.title}"
        }
        img {
            src = "https://via.placeholder.com/640x360.png?text=Video+Player+Placeholder"
        }
    }
}
```

2. Because the VideoPlayerProps interface specifies that the VideoPlayer component takes a non-null Video, make sure to handle this in the App component accordingly.

   In App.kt, replace the previous div snippet for the video player with the following:

```
currentVideo?.let { curr ->
    VideoPlayer {
        video = curr
    }
}
```

   The let scope function ensures that the VideoPlayer component is only added when state.currentVideo is not null.

Now clicking an entry in the list will bring up the video player and populate it with the information from the clicked entry.

Add a button and wire it

To make it possible for users to mark a video as watched or unwatched and to move it between the two lists, add a button to the VideoPlayer component.

Since this button will move videos between two different lists, the logic handling state change needs to be lifted out of the VideoPlayer and passed in from the parent as a prop. The button should look different based on whether the video has been watched or not. This is also information you need to pass as a prop.

1. Expand the VideoPlayerProps interface in VideoPlayer.kt to include properties for those two cases:

```
external interface VideoPlayerProps : Props {
    var video: Video
    var onWatchedButtonPressed: (Video) -> Unit
    var unwatchedVideo: Boolean
}
```

2. You can now add the button to the actual component. Copy the following snippet into the body of the VideoPlayer component, between the h3 and img tags:

```
button {
    css {
        display = Display.block
        backgroundColor = if (props.unwatchedVideo) NamedColor.lightgreen else NamedColor.red
    }
    onClick = {
        props.onWatchedButtonPressed(props.video)
    }
    if (props.unwatchedVideo) {
        +"Mark as watched"
    } else {
        +"Mark as unwatched"
    }
}
```

   With the help of Kotlin CSS DSL that make it possible to change styles dynamically, you can change the color of the button using a basic Kotlin if expression.

Move video lists to the application state

Now it's time to adjust the VideoPlayer usage site in the App component. When the button is clicked, a video should be moved from the unwatched list to the watched list or vice versa. Since these lists can now actually change, move them into the application state:

1. In App.kt, add the following useState() calls to the top of the App component:

```
val App = FC<Props> {
    var currentVideo: Video? by useState(null)
    var unwatchedVideos: List<Video> by useState(listOf(
        Video(1, "Opening Keynote", "Andrey Breslav", "https://youtu.be/PsaFVLr8t4E"),
        Video(2, "Dissecting the stdlib", "Huyen Tue Dao", "https://youtu.be/Fzt_9I733Yg"),
        Video(3, "Kotlin and Spring Boot", "Nicolas Frankel", "https://youtu.be/pSiZVAeReeg")
    ))
    var watchedVideos: List<Video> by useState(listOf(
        Video(4, "Creating Internal DSLs in Kotlin", "Venkat Subramaniam", "https://youtu.be/JzTeAM8N1-o")
    ))

    // . . .
}
```

2. Since all the demo data is included in the default values for watchedVideos and unwatchedVideos directly, you no longer need the file-level declarations. In Main.kt, delete the declarations for watchedVideos and unwatchedVideos.

3. Change the call-site for VideoPlayer in the App component that belongs to the video player to look like this:

```
VideoPlayer {
    video = curr
    unwatchedVideo = curr in unwatchedVideos
    onWatchedButtonPressed = {
        if (video in unwatchedVideos) {
            unwatchedVideos = unwatchedVideos - video
            watchedVideos = watchedVideos + video
        } else {
            watchedVideos = watchedVideos - video
            unwatchedVideos = unwatchedVideos + video
        }
    }
}
```

Go back to the browser, select a video, and press the button a few times. The video will jump between the two lists.

Use packages from npm

To make the app usable, you still need a video player that actually plays videos and some buttons to help people share the content.

React has a rich ecosystem with a lot of pre-made components you can use instead of building this functionality yourself.

Add the video player component

To replace the placeholder video component with an actual YouTube player, use the react-player package from npm. It can play videos and allows you to control the appearance of the player.

For the component documentation and the API description, see its README in GitHub.

1. Check the build.gradle.kts file. The react-player package should be already included:

```
dependencies {
    // ...
    // Video Player
    implementation(npm("react-player", "2.12.0"))
    // ...
}
```

   As you can see, npm dependencies can be added to a Kotlin/JS project by using the npm() function in the dependencies block of the build file. The Gradle plugin then takes care of downloading and installing these dependencies for you. To do so, it uses its own bundled installation of the yarn package manager.

2. To use the JavaScript package from inside the React application, it's necessary to tell the Kotlin compiler what to expect by providing it with external declarations.

   Create a new ReactYouTube.kt file and add the following content:

```
@file:JsModule("react-player")
@file:JsNonModule
```

```kotlin
import react.*

@JsName("default")
external val ReactPlayer: ComponentClass<dynamic>
```

When the compiler sees an external declaration like ReactPlayer, it assumes that the implementation for the corresponding class is provided by the dependency and doesn't generate code for it.

The last two lines are equivalent to a JavaScript import like require("react-player").default;. They tell the compiler that it's certain that a component will conform to ComponentClass<dynamic> at runtime.

However, in this configuration, the generic type for the props accepted by ReactPlayer is set to dynamic. That means the compiler will accept any code, at the risk of breaking things at runtime.

A better alternative would be to create an external interface that specifies what kind of properties belong to the props for this external component. You can learn about the props' interface in the README for the component. In this case, use the url and controls props:

1. Adjust the content of ReactYouTube.kt accordingly:

```kotlin
@file:JsModule("react-player")
@file:JsNonModule

import react.*

@JsName("default")
external val ReactPlayer: ComponentClass<ReactPlayerProps>

external interface ReactPlayerProps : Props {
    var url: String
    var controls: Boolean
}
```

2. You can now use the new ReactPlayer to replace the gray placeholder rectangle in the VideoPlayer component. In VideoPlayer.kt, replace the img tag with the following snippet:

```kotlin
ReactPlayer {
    url = props.video.videoUrl
    controls = true
}
```

Add social share buttons

An easy way to share the application's content is to have social share buttons for messengers and email. You can use an off-the-shelf React component for this as well, for example, react-share:

1. Check the build.gradle.kts file. This npm library should already be included:

```kotlin
dependencies {
    // ...
    // Share Buttons
    implementation(npm("react-share", "4.4.1"))
    // ...
}
```

2. To use react-share from Kotlin, you'll need to write more basic external declarations. The examples on GitHub show that a share button consists of two React components: EmailShareButton and EmailIcon, for example. Different types of share buttons and icons all have the same kind of interface. You'll create the external declarations for each component the same way you already did for the video player.

   Add the following code to a new ReactShare.kt file:

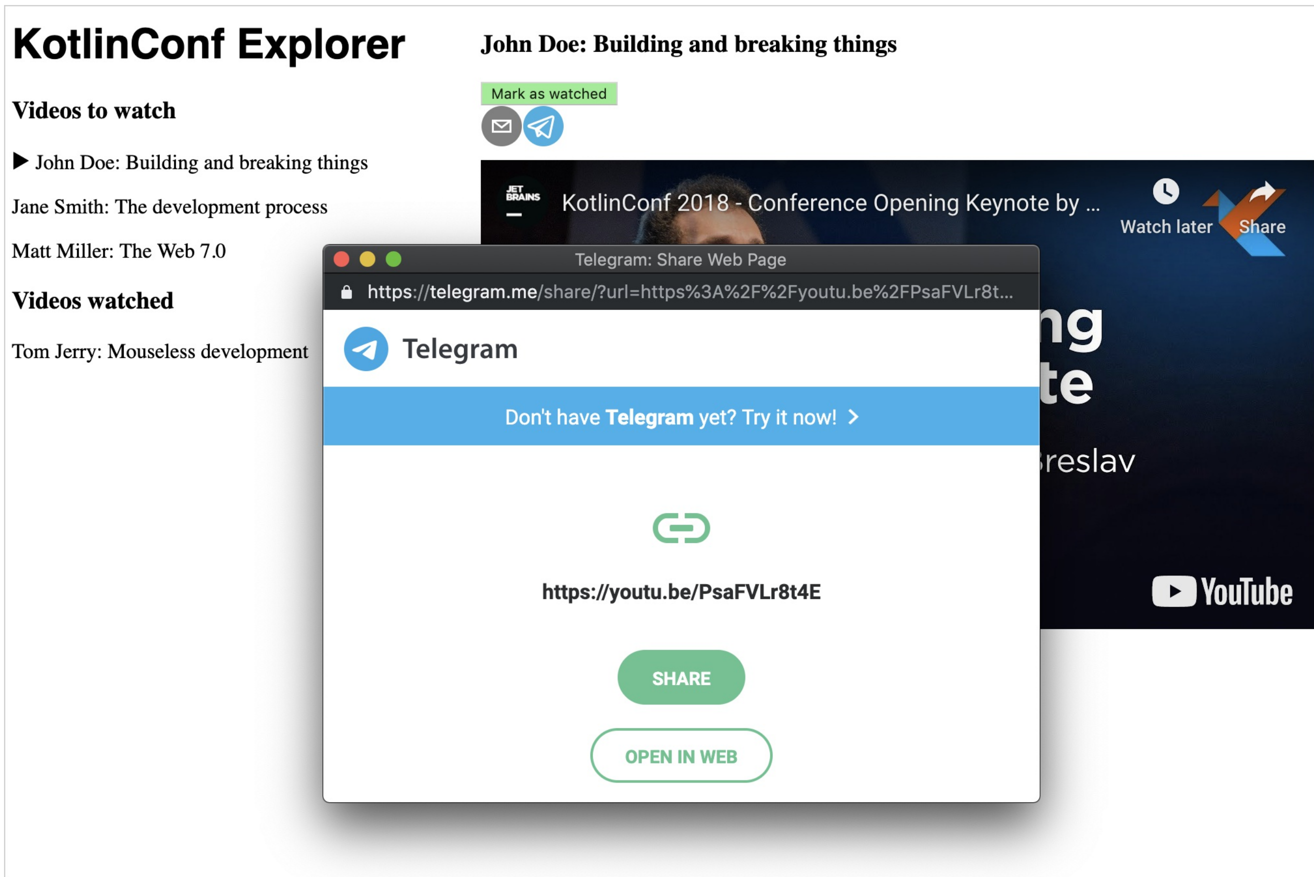
```kotlin
@file:JsModule("react-share")
@file:JsNonModule

import react.ComponentClass
import react.Props

@JsName("EmailIcon")
external val EmailIcon: ComponentClass<IconProps>
```

```
@JsName("EmailShareButton")
external val EmailShareButton: ComponentClass<ShareButtonProps>

@JsName("TelegramIcon")
external val TelegramIcon: ComponentClass<IconProps>

@JsName("TelegramShareButton")
external val TelegramShareButton: ComponentClass<ShareButtonProps>

external interface ShareButtonProps : Props {
    var url: String
}

external interface IconProps : Props {
    var size: Int
    var round: Boolean
}
```

3. Add new components into the user interface of the application. In VideoPlayer.kt, add two share buttons in a div right above the usage of ReactPlayer:

```
// . . .

div {
    css {
         position = Position.absolute
         top = 10.px
         right = 10.px
     }
    EmailShareButton {
        url = props.video.videoUrl
        EmailIcon {
            size = 32
            round = true
        }
    }
    TelegramShareButton {
        url = props.video.videoUrl
        TelegramIcon {
            size = 32
            round = true
        }
    }
}

// . . .
```

You can now check your browser and see whether the buttons actually work. When clicking on the button, a share window should appear with the URL of the video. If the buttons don't show up or work, you may need to disable your ad and social media blocker.



Share window

Feel free to repeat this step with share buttons for other social networks available in react-share.

Use an external REST API

You can now replace the hard-coded demo data with some real data from a REST API in the app.

For this tutorial, there's a small API. It offers only a single endpoint, videos, and takes a numeric parameter to access an element from the list. If you visit the API with your browser, you will see that the objects returned from the API have the same structure as Video objects.

Use JS functionality from Kotlin

Browsers already come with a large variety of Web APIs. You can also use them from Kotlin/JS, since it includes wrappers for these APIs out of the box. One example is the fetch API, which is used for making HTTP requests.

The first potential issue is that browser APIs like fetch() use callbacks to perform non-blocking operations. When multiple callbacks are supposed to run one after the other, they need to be nested. Naturally, the code gets heavily indented, with more and more pieces of functionality stacked inside each other, which makes it harder to read.

To overcome this, you can use Kotlin's coroutines, a better approach for such functionality.

The second issue arises from the dynamically typed nature of JavaScript. There are no guarantees about the type of data returned from the external API. To solve this, you can use the kotlinx.serialization library.

Check the build.gradle.kts file. The relevant snippet should already exist:

```
dependencies {
    // . . .

    // Coroutines & serialization
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4")
}
```

Add serialization

When you call an external API, you get back JSON-formatted text that still needs to be turned into a Kotlin object that can be worked with.

kotlinx.serialization is a library that makes it possible to write these types of conversions from JSON strings to Kotlin objects.

1. Check the build.gradle.kts file. The corresponding snippet should already exist:

```
plugins {
    // . . .
    kotlin("plugin.serialization") version "1.9.0"
}

dependencies {
    // . . .

    // Serialization
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.0")
}
```

2. As preparation for fetching the first video, it's necessary to tell the serialization library about the Video class. In Main.kt, add the @Serializable annotation to its definition:

```
@Serializable
data class Video(
    val id: Int,
    val title: String,
    val speaker: String,
    val videoUrl: String
)
```

Fetch videos

To fetch a video from the API, add the following function in App.kt (or a new file):

```
suspend fun fetchVideo(id: Int): Video {
    val response = window
        .fetch("https://my-json-server.typicode.com/kotlin-hands-on/kotlinconf-json/videos/$id")
        .await()
        .text()
        .await()
    return Json.decodeFromString(response)
}
```

- Suspending function fetch()es a video with a given id from the API. This response may take a while, so you await() the result. Next, text(), which uses a callback, reads the body from the response. Then you await() its completion.
- Before returning the value of the function, you pass it to Json.decodeFromString, a function from kotlinx.coroutines. It converts the JSON text you received from the request into a Kotlin object with the appropriate fields.
- The window.fetch function call returns a Promise object. You normally would have to define a callback handler that gets invoked once the Promise is resolved and a result is available. However, with coroutines, you can await() those promises. Whenever a function like await() is called, the method stops (suspends) its execution. Its execution continues once the Promise can be resolved.

To give users a selection of videos, define the fetchVideos() function, which will fetch 25 videos from the same API as above. To run all the requests concurrently, use the async functionality provided by Kotlin's coroutines:

1. Add the following implementation to your App.kt:

```
suspend fun fetchVideos(): List<Video> = coroutineScope {
    (1..25).map { id ->
        async {
            fetchVideo(id)
        }
    }.awaitAll()
}
```

Following the principle of structured concurrency, the implementation is wrapped in a coroutineScope. You can then start 25 asynchronous tasks (one per

request) and wait for all of them to complete.

2. You can now add data to your application. Add the definition for a mainScope, and change your App component so it starts with the following snippet. Don't forget to replace demo values with emptyLists instances as well:

```
val mainScope = MainScope()

val App = FC<Props> {
    var currentVideo: Video? by useState(null)
    var unwatchedVideos: List<Video> by useState(emptyList())
    var watchedVideos: List<Video> by useState(emptyList())

    useEffectOnce {
        mainScope.launch {
            unwatchedVideos = fetchVideos()
        }
    }

// . . .
```

- The MainScope() is a part of Kotlin's structured concurrency model and creates the scope for asynchronous tasks to run in.
- useEffectOnce is another React hook (specifically, a simplified version of the useEffect hook). It indicates that the component performs a side effect. It doesn't just render itself but also communicates over the network.

Check your browser. The application should show actual data:

KotlinConf Explorer

Videos to watch

Romain Guy: Graphics Programming with Kotlin

Nicolas Frankel: Kotlin and Spring Boot, a Match Made in Heaven

Dmitry Kandalov: Live Coding Kotlin/Native Snake

Marvin Ramin: Safe(r) Kotlin Code - Static Analysis Tools for Kotlin

Chris Banes: Android Suspenders

Annyce Davis: GraphQL Powered by Kotlin

Thomas Nield: Mathematical Modeling with Kotlin

Zac Sweers: Annotation Processing in a Kotlin World

Felipe Lima: Architecting a Kotlin JVM and JS Multiplatform Project

Uberto Barbini: Functional CQRS in Kotlin

▶ Holger Brandl: Building Data Science Workflows with Kotlin

Nat Pryce: Exploring the Kotlin Type Hierarchy from Top to Bottom

Nikolay Igotti: Kotlin/Native Concurrency Model

Kevin Most: Writing Your First Kotlin Compiler Plugin

David Wursteisen: Beat the High-Score: Build a Game Using libGDX and Kotlin

Florina: Shaping Your App's Architecture with Kotlin and Architecture Components

Svetlana Isakova: New Type Inference and Related Language Features

Ivan Sanchez & David Denton: Server as a Function in Kotlin

Holger Brandl: Building Data Science Workflows with Kotlin

Mark as watched

Fetched data from API

When you load the page:

- The code of the App component will be invoked. This starts the code in the useEffectOnce block.
- The App component is rendered with empty lists for the watched and unwatched videos.
- When the API requests finish, the useEffectOnce block assigns it to the App component's state. This triggers a re-render.
- The code of the App component will be invoked again, but the useEffectOnce block will not run for a second time.

If you want to get an in-depth understanding of how coroutines work, check out this tutorial on coroutines.

Deploy to production and the cloud

It's time to get the application published to the cloud and make it accessible to other people.

Package a production build

To package all assets in production mode, run the build task in Gradle via the tool window in IntelliJ IDEA or by running ./gradlew build. This generates an optimized project build, applying various improvements such as DCE (dead code elimination).

Once the build has finished, you can find all the files needed for deployment in /build/dist. They include the JavaScript files, HTML files, and other resources required to run the application. You can put them on a static HTTP server, serve them using GitHub Pages, or host them on a cloud provider of your choice.

Deploy to Heroku

Heroku makes it quite simple to spin up an application that is reachable under its own domain. Their free tier should be sufficient for development purposes.

1. Create an account.
2. Install and authenticate the CLI client.
3. Create a Git repository and attach a Heroku app by running the following commands in the Terminal while in the project root:

```
git init
heroku create
git add .
git commit -m "initial commit"
```

4. Unlike a regular JVM application that would run on Heroku (one written with Ktor or Spring Boot, for example), your app generates static HTML pages and JavaScript files that need to be served accordingly. You can adjust the required buildpacks to serve the program properly:

```
heroku buildpacks:set heroku/gradle
heroku buildpacks:add https://github.com/heroku/heroku-buildpack-static.git
```

5. To allow the heroku/gradle buildpack to run properly, a stage task needs to be in the build.gradle.kts file. This task is equivalent to the build task, and the corresponding alias is already included at the bottom of the file:

```
// Heroku Deployment
tasks.register("stage") {
    dependsOn("build")
}
```

6. Add a new static.json file to the project root to configure the buildpack-static.
7. Add the root property inside the file:

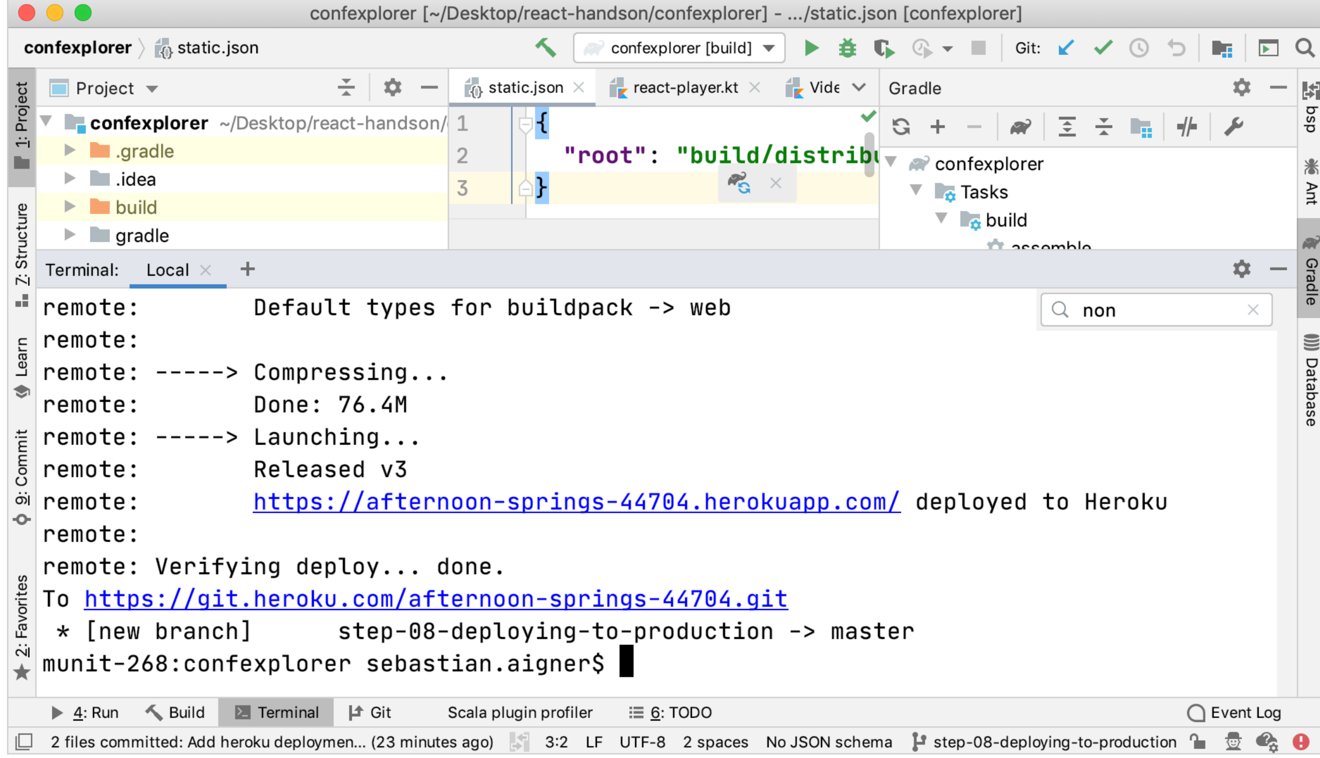
```
{
    "root": "build/distributions"
}
```

8. You can now trigger a deployment, for example, by running the following command:

```
git add -A
git commit -m "add stage task and static content root configuration"
git push heroku master
```

> If you're pushing from a non-main branch, adjust the command to push to the main remote, for example, git push heroku feature-branch:main.

If the deployment is successful, you will see the URL people can use to reach the application on the internet.



Web app deployment to production

You can find this state of the project on the finished branch here.

What's next

Add more features

You can use the resulting app as a jumping-off point to explore more advanced topics in the realm of React, Kotlin/JS, and more.

- Search. You can add a search field to filter the list of talks – by title or by author, for example. Learn about how HTML form elements work in React.
- Persistence. Currently, the application loses track of the viewer's watch list every time the page gets reloaded. Consider building your own backend, using one of the web frameworks available for Kotlin (such as Ktor). Alternatively, look into ways to store information on the client.
- Complex APIs. Lots of datasets and APIs are available. You can pull all sorts of data into your application. For example, you can build a visualizer for cat photos or a royalty-free stock photo API.

Improve the style: responsiveness and grids

The application design is still very simple and won't look great on mobile devices or in narrow windows. Explore more of the CSS DSL to make the app more accessible.

Join the community and get help

The best way to report problems and get help is the kotlin-wrappers issue tracker. If you can't find a ticket for your issue, feel free to file a new one. You can also join the official Kotlin Slack. There are channels for #javascript and #react.

Learn more about coroutines

If you're interested in finding out more about how you can write concurrent code, check out the tutorial on coroutines.

Learn more about React

Now that you know the basic React concepts and how they translate to Kotlin, you can convert some other concepts outlined in the official guides on React into Kotlin.

Get started with Kotlin custom scripting – tutorial

Kotlin scripting is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin scripting is the technology that enables executing Kotlin code as scripts without prior compilation or packaging into executables.

For an overview of Kotlin scripting with examples, check out the talk Implementing the Gradle Kotlin DSL by Rodrigo Oliveira from KotlinConf'19.

In this tutorial, you'll create a Kotlin scripting project that executes arbitrary Kotlin code with Maven dependencies. You'll be able to execute scripts like this:

```
@file:Repository("https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven")
@file:DependsOn("org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3")

import kotlinx.html.*
import kotlinx.html.stream.*
import kotlinx.html.attributes.*

val addressee = "World"

print(
    createHTML().html {
        body {
            h1 { +"Hello, $addressee!" }
        }
    }
)
```

The specified Maven dependency (kotlinx-html-jvm for this example) will be resolved from the specified Maven repository or local cache during execution and used for the rest of the script.

Project structure

A minimal Kotlin custom scripting project contains two parts:

- Script definition – a set of parameters and configurations that define how this script type should be recognized, handled, compiled, and executed.
- Scripting host – an application or component that handles script compilation and execution – actually running scripts of this type.

With all of this in mind, it's best to split the project into two modules.

Before you start

Download and install the latest version of IntelliJ IDEA.

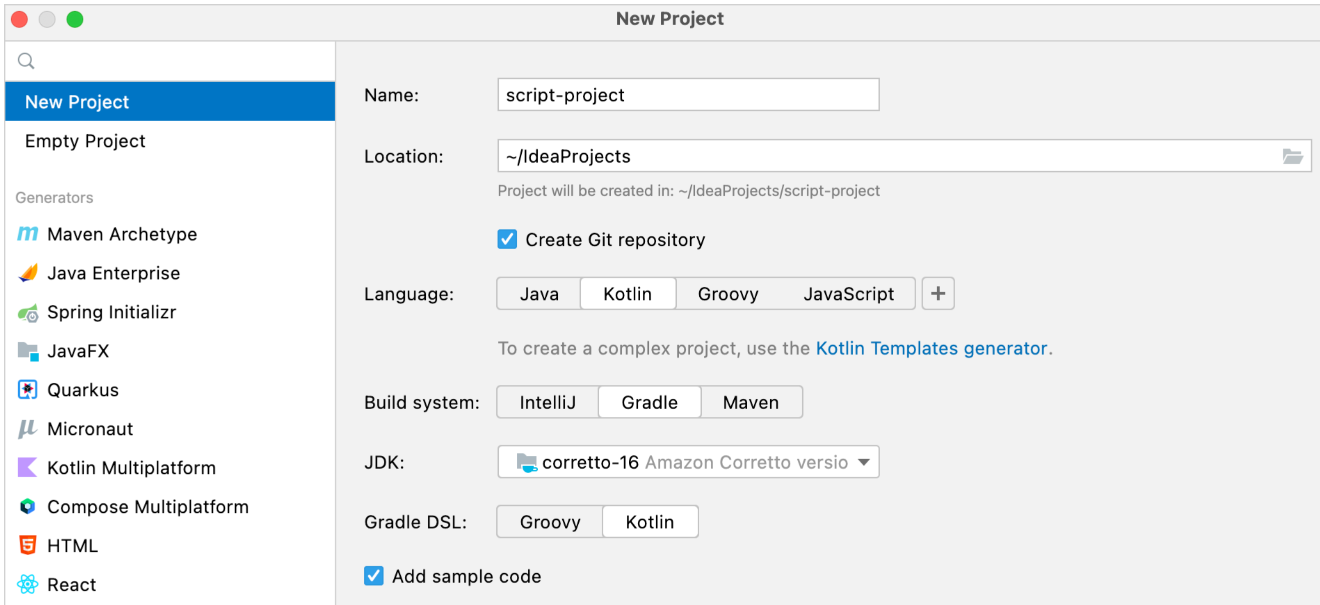
Create a project

1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select New Project.
3. Name the new project and change its location if necessary.

   Select the Create Git repository checkbox to place the new project under version control. You will be able to do it later at any time.

4. From the Language list, select Kotlin.

5. Select the Gradle build system.
6. From the JDK list, select the JDK that you want to use in your project.
   * If the JDK is installed on your computer, but not defined in the IDE, select Add JDK and specify the path to the JDK home directory.
   * If you don't have the necessary JDK on your computer, select Download JDK.
7. Select the Kotlin or Gradle language for the Gradle DSL.
8. Click Create.

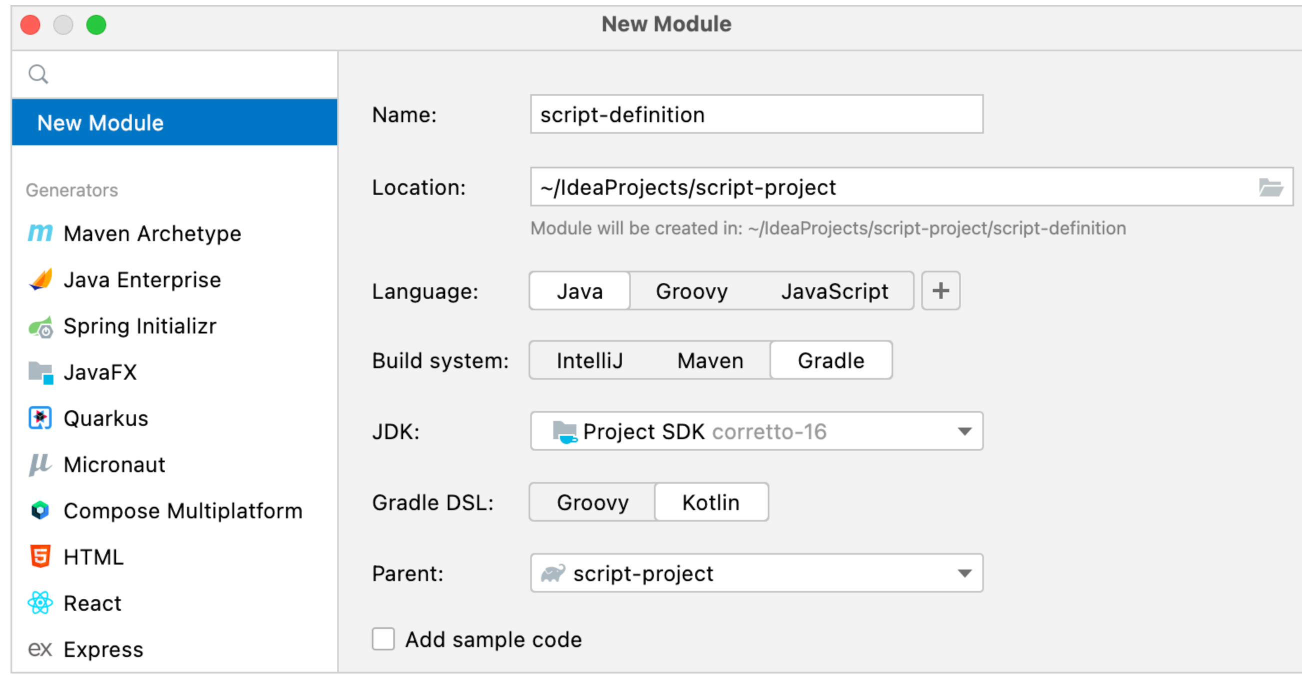


Create a root project for custom Kotlin scripting

Add scripting modules

Now you have an empty Kotlin/JVM Gradle project. Add the required modules, script definition and scripting host:

1. In IntelliJ IDEA, select File | New | Module.
2. In the panel on the left, select New Module. This module will be the script definition.
3. Name the new module and change its location if necessary.
4. From the Language list, select Java.
5. Select the Gradle build system and Kotlin for the Gradle DSL if you want to write the build script in Kotlin.
6. As a module's parent, select the root module.
7. Click Create.

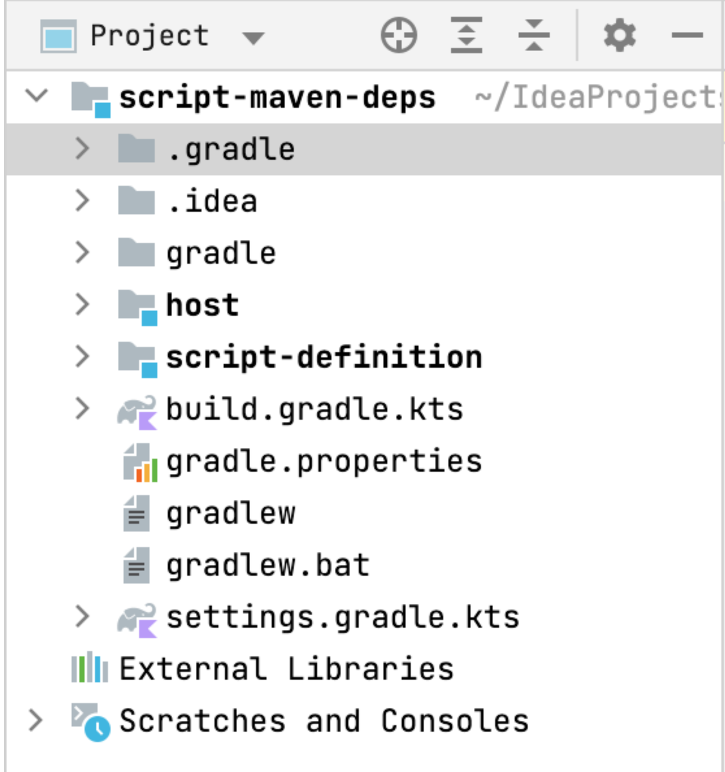


Create script definition module

8. In the module's build.gradle(.kts) file, remove the version of the Kotlin Gradle plugin. It is already in the root project's build script.

9. Repeat previous steps one more time to create a module for the scripting host.

The project should have the following structure:



Custom scripting project structure

You can find an example of such a project and more Kotlin scripting examples in the kotlin-script-examples GitHub repository.

Create a script definition

First, define the script type: what developers can write in scripts of this type and how it will be handled. In this tutorial, this includes support for the @Repository and @DependsOn annotations in the scripts.

1. In the script definition module, add the dependencies on the Kotlin scripting components in the dependencies block of build.gradle(.kts). These dependencies provide the APIs you will need for the script definition:

   Kotlin

   ```
   dependencies {
       implementation("org.jetbrains.kotlin:kotlin-scripting-common")
       implementation("org.jetbrains.kotlin:kotlin-scripting-jvm")
       implementation("org.jetbrains.kotlin:kotlin-scripting-dependencies")
       implementation("org.jetbrains.kotlin:kotlin-scripting-dependencies-maven")
       // coroutines dependency is required for this particular definition
       implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
   }
   ```

   Groovy

   ```
   dependencies {
       implementation 'org.jetbrains.kotlin:kotlin-scripting-common'
       implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm'
       implementation 'org.jetbrains.kotlin:kotlin-scripting-dependencies'
       implementation 'org.jetbrains.kotlin:kotlin-scripting-dependencies-maven'
       // coroutines dependency is required for this particular definition
       implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.7.1'

   }
   ```

2. Create the src/main/kotlin/ directory in the module and add a Kotlin source file, for example, scriptDef.kt.

3. In scriptDef.kt, create a class. It will be a superclass for scripts of this type, so declare it abstract or open.

   ```
   // abstract (or open) superclass for scripts of this type
   abstract class ScriptWithMavenDeps
   ```

   This class will also serve as a reference to the script definition later.

4. To make the class a script definition, mark it with the @KotlinScript annotation. Pass two parameters to the annotation:

   - fileExtension – a string ending with .kts that defines a file extension for scripts of this type.
   - compilationConfiguration – a Kotlin class that extends ScriptCompilationConfiguration and defines the compilation specifics for this script definition. You'll create it in the next step.

   ```
   // @KotlinScript annotation marks a script definition class
   @KotlinScript(
       // File extension for the script type
       fileExtension = "scriptwithdeps.kts",
       // Compilation configuration for the script type
       compilationConfiguration = ScriptWithMavenDepsConfiguration::class
   )
   abstract class ScriptWithMavenDeps

   object ScriptWithMavenDepsConfiguration: ScriptCompilationConfiguration()
   ```

   > In this tutorial, we provide only the working code without explaining Kotlin scripting API. You can find the same code with a detailed explanation on GitHub.

5. Define the script compilation configuration as shown below.

   ```
   object ScriptWithMavenDepsConfiguration : ScriptCompilationConfiguration(
       {
   ```

```kotlin
        // Implicit imports for all scripts of this type
        defaultImports(DependsOn::class, Repository::class)
        jvm {
            // Extract the whole classpath from context classloader and use it as dependencies
            dependenciesFromCurrentContext(wholeClasspath = true)
        }
        // Callbacks
        refineConfiguration {
            // Process specified annotations with the provided handler
            onAnnotations(DependsOn::class, Repository::class, handler = ::configureMavenDepsOnAnnotations)
        }
    }
)
```

The configureMavenDepsOnAnnotations function is as follows:

```kotlin
// Handler that reconfigures the compilation on the fly
fun configureMavenDepsOnAnnotations(context: ScriptConfigurationRefinementContext):
ResultWithDiagnostics<ScriptCompilationConfiguration> {
    val annotations = context.collectedData?.get(ScriptCollectedData.collectedAnnotations)?.takeIf { it.isNotEmpty() }
        ?: return context.compilationConfiguration.asSuccess()
    return runBlocking {
        resolver.resolveFromScriptSourceAnnotations(annotations)
    }.onSuccess {
        context.compilationConfiguration.with {
            dependencies.append(JvmDependency(it))
        }.asSuccess()
    }
}

private val resolver = CompoundDependenciesResolver(FileSystemDependenciesResolver(), MavenDependenciesResolver())
```

You can find the full code here.

Create a scripting host

The next step is creating the scripting host – the component that handles the script execution.

1. In the scripting host module, add the dependencies in the dependencies block of build.gradle(.kts):

   - Kotlin scripting components that provide the APIs you need for the scripting host
   - The script definition module you created previously

   Kotlin

   ```kotlin
   dependencies {
       implementation("org.jetbrains.kotlin:kotlin-scripting-common")
       implementation("org.jetbrains.kotlin:kotlin-scripting-jvm")
       implementation("org.jetbrains.kotlin:kotlin-scripting-jvm-host")
       implementation(project(":script-definition")) // the script definition module
   }
   ```

   Groovy

   ```groovy
   dependencies {
       implementation 'org.jetbrains.kotlin:kotlin-scripting-common'
       implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm'
       implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm-host'
       implementation project(':script-definition') // the script definition module
   }
   ```

2. Create the src/main/kotlin/ directory in the module and add a Kotlin source file, for example, host.kt.

3. Define the main function for the application. In its body, check that it has one argument – the path to the script file – and execute the script. You'll define the script execution in a separate function evalFile in the next step. Declare it empty for now.

   main can look like this:

```
fun main(vararg args: String) {
    if (args.size != 1) {
        println("usage: <app> <script file>")
    } else {
        val scriptFile = File(args[0])
        println("Executing script $scriptFile")
        evalFile(scriptFile)
    }
}
```

4. Define the script evaluation function. This is where you'll use the script definition. Obtain it by calling createJvmCompilationConfigurationFromTemplate with the script definition class as a type parameter. Then call BasicJvmScriptingHost().eval, passing it the script code and its compilation configuration. eval returns an instance of ResultWithDiagnostics, so set it as your function's return type.

```
fun evalFile(scriptFile: File): ResultWithDiagnostics<EvaluationResult> {
    val compilationConfiguration = createJvmCompilationConfigurationFromTemplate<ScriptWithMavenDeps>()
    return BasicJvmScriptingHost().eval(scriptFile.toScriptSource(), compilationConfiguration, null)
}
```

5. Adjust the main function to print information about the script execution:

```
fun main(vararg args: String) {
    if (args.size != 1) {
        println("usage: <app> <script file>")
    } else {
        val scriptFile = File(args[0])
        println("Executing script $scriptFile")
        val res = evalFile(scriptFile)
        res.reports.forEach {
            if (it.severity > ScriptDiagnostic.Severity.DEBUG) {
                println(" : ${it.message}" + if (it.exception == null) "" else ": ${it.exception}")
            }
        }
    }
}
```

You can find the full code here

Run scripts

To check how your scripting host works, prepare a script to execute and a run configuration.

1. Create the file html.scriptwithdeps.kts with the following content in the project root directory:

```
@file:Repository("https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven")
@file:DependsOn("org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3")

import kotlinx.html.*; import kotlinx.html.stream.*; import kotlinx.html.attributes.*

val addressee = "World"

print(
    createHTML().html {
        body {
            h1 { +"Hello, $addressee!" }
        }
    }
)
```

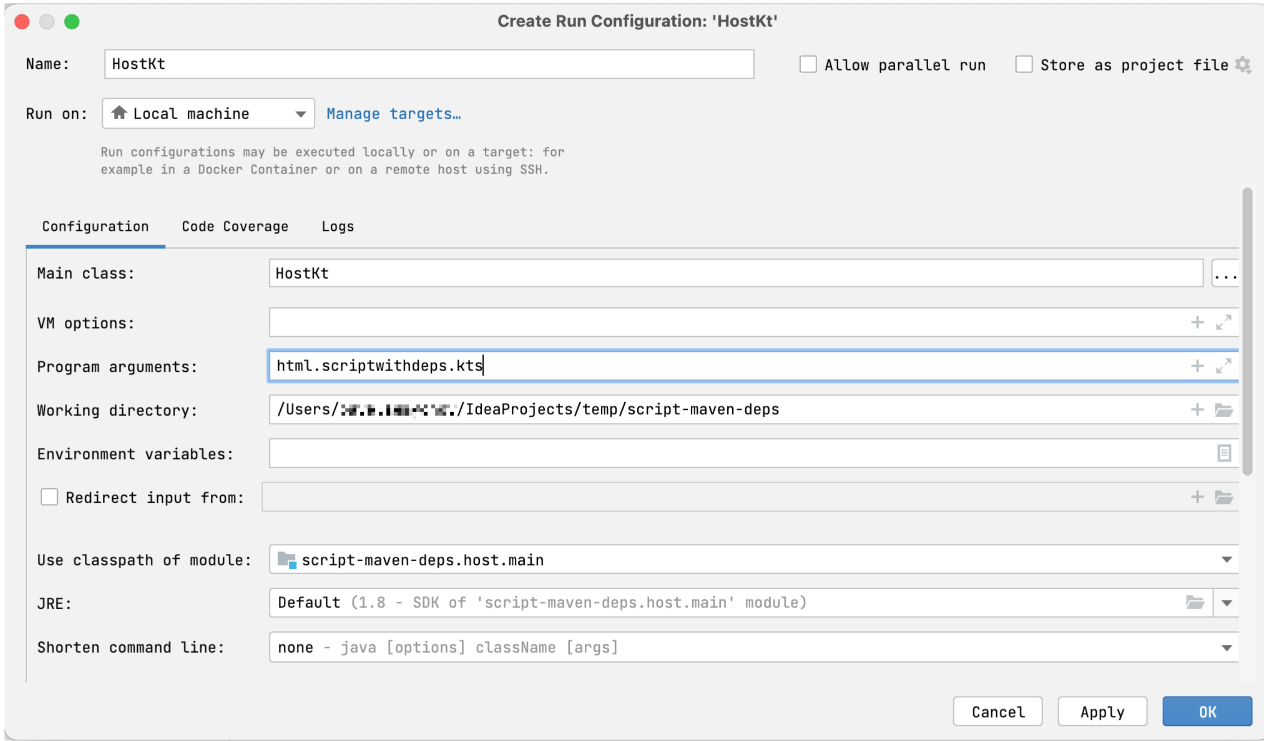
It uses functions from the kotlinx-html-jvm library which is referenced in the @DependsOn annotation argument.

2. Create a run configuration that starts the scripting host and executes this file:

   1. Open host.kt and navigate to the main function. It has a Run gutter icon on the left.

   2. Right-click the gutter icon and select Modify Run Configuration.

   3. In the Create Run Configuration dialog, add the script file name to Program arguments and click OK.



Scripting host run configuration

3. Run the created configuration.

You'll see how the script is executed, resolving the dependency on kotlinx-html-jvm in the specified repository and printing the results of calling its functions:

```
<html>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

Resolving dependencies may take some time on the first run. Subsequent runs will complete much faster because they use downloaded dependencies from the local Maven repository.

What's next?

Once you've created a simple Kotlin scripting project, find more information on this topic:

- Read the Kotlin scripting KEEP
- Browse more Kotlin scripting examples
- Watch the talk Implementing the Gradle Kotlin DSL by Rodrigo Oliveira

Collections overview

The Kotlin Standard Library provides a comprehensive set of tools for managing collections – groups of a variable number of items (possibly zero) that are significant to the problem being solved and are commonly operated on.

Collections are a common concept for most programming languages, so if you're familiar with, for example, Java or Python collections, you can skip this introduction and proceed to the detailed sections.

A collection usually contains a number of objects (this number may also be zero) of the same type. Objects in a collection are called elements or items. For example, all the students in a department form a collection that can be used to calculate their average age.

The following collection types are relevant for Kotlin:

- List is an ordered collection with access to elements by indices – integer numbers that reflect their position. Elements can occur more than once in a list. An example of a list is a telephone number: it's a group of digits, their order is important, and they can repeat.
- Set is a collection of unique elements. It reflects the mathematical abstraction of set: a group of objects without repetitions. Generally, the order of set elements has no significance. For example, the numbers on lottery tickets form a set: they are unique, and their order is not important.
- Map (or dictionary) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value. The values can be duplicates. Maps are useful for storing logical connections between objects, for example, an employee's ID and their position.

Kotlin lets you manipulate collections independently of the exact type of objects stored in them. In other words, you add a String to a list of Strings the same way as you would do with Ints or a user-defined class. So, the Kotlin Standard Library offers generic interfaces, classes, and functions for creating, populating, and managing collections of any type.

The collection interfaces and related functions are located in the kotlin.collections package. Let's get an overview of its contents.

Collection types

The Kotlin Standard Library provides implementations for basic collection types: sets, lists, and maps. A pair of interfaces represent each collection type:

- A read-only interface that provides operations for accessing collection elements.
- A mutable interface that extends the corresponding read-only interface with write operations: adding, removing, and updating its elements.

Note that altering a mutable collection doesn't require it to be a var: write operations modify the same mutable collection object, so the reference doesn't change. Although, if you try to reassign a val collection, you'll get a compilation error.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.add("five")   // this is OK
    println(numbers)
    //numbers = mutableListOf("six", "seven")      // compilation error
//sampleEnd

}
```

The read-only collection types are covariant. This means that, if a Rectangle class inherits from Shape, you can use a List<Rectangle> anywhere the List<Shape> is required. In other words, the collection types have the same subtyping relationship as the element types. Maps are covariant on the value type, but not on the key type.

In turn, mutable collections aren't covariant; otherwise, this would lead to runtime failures. If MutableList<Rectangle> was a subtype of MutableList<Shape>, you could insert other Shape inheritors (for example, Circle) into it, thus violating its Rectangle type argument.

Below is a diagram of the Kotlin collection interfaces:

Collection interfaces hierarchy

Let's walk through the interfaces and their implementations. To learn about Collection, read the section below. To learn about List, Set, and Map, you can either read the corresponding sections or watch a video by Sebastian Aigner, Kotlin Developer Advocate:

Watch video online.

Collection

Collection<T> is the root of the collection hierarchy. This interface represents the common behavior of a read-only collection: retrieving size, checking item

membership, and so on. Collection inherits from the Iterable<T> interface that defines the operations for iterating elements. You can use Collection as a parameter of a function that applies to different collection types. For more specific cases, use the Collection's inheritors: List and Set.

```
fun printAll(strings: Collection<String>) {
    for(s in strings) print("$s ")
    println()
}

fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet)
}
```

MutableCollection<T> is a Collection with write operations, such as add and remove.

```
fun List<String>.getShortWordsTo(shortWords: MutableList<String>, maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // throwing away the articles
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split(" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords)
}
```

List

List<T> stores elements in a specified order and provides indexed access to them. Indices start from zero – the index of the first element – and go to lastIndex which is the (list.size - 1).

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println("Number of elements: ${numbers.size}")
    println("Third element: ${numbers.get(2)}")
    println("Fourth element: ${numbers[3]}")
    println("Index of element \"two\" ${numbers.indexOf("two")}")
//sampleEnd
}
```

List elements (including nulls) can duplicate: a list can contain any number of equal objects or occurrences of a single object. Two lists are considered equal if they have the same sizes and structurally equal elements at the same positions.

```
data class Person(var name: String, var age: Int)

fun main() {
//sampleStart
    val bob = Person("Bob", 31)
    val people = listOf(Person("Adam", 20), bob, bob)
    val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
    println(people == people2)
    bob.age = 32
    println(people == people2)
//sampleEnd
}
```

MutableList<T> is a List with list-specific write operations, for example, to add or remove an element at a specific position.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    numbers.removeAt(1)
    numbers[0] = 0
    numbers.shuffle()
```

```
    println(numbers)
//sampleEnd
}
```

As you see, in some aspects lists are very similar to arrays. However, there is one important difference: an array's size is defined upon initialization and is never changed; in turn, a list doesn't have a predefined size; a list's size can be changed as a result of write operations: adding, updating, or removing elements.

In Kotlin, the default implementation of MutableList is ArrayList which you can think of as a resizable array.

Set

Set<T> stores unique elements; their order is generally undefined. null elements are unique as well: a Set can contain only one null. Two sets are equal if they have the same size, and for each element of a set there is an equal element in the other set.

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3, 4)
    println("Number of elements: ${numbers.size}")
    if (numbers.contains(1)) println("1 is in the set")

    val numbersBackwards = setOf(4, 3, 2, 1)
    println("The sets are equal: ${numbers == numbersBackwards}")
//sampleEnd
}
```

MutableSet is a Set with write operations from MutableCollection.

The default implementation of MutableSet – LinkedHashSet – preserves the order of elements insertion. Hence, the functions that rely on the order, such as first() or last(), return predictable results on such sets.

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet is the default implementation
    val numbersBackwards = setOf(4, 3, 2, 1)

    println(numbers.first() == numbersBackwards.first())
    println(numbers.first() == numbersBackwards.last())
//sampleEnd
}
```

An alternative implementation – HashSet – says nothing about the elements order, so calling such functions on it returns unpredictable results. However, HashSet requires less memory to store the same number of elements.

Map

Map<K, V> is not an inheritor of the Collection interface; however, it's a Kotlin collection type as well. A Map stores key-value pairs (or entries); keys are unique, but different keys can be paired with equal values. The Map interface provides specific functions, such as access to value by key, searching keys and values, and so on.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

    println("All keys: ${numbersMap.keys}")
    println("All values: ${numbersMap.values}")
    if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
    if (1 in numbersMap.values) println("The value 1 is in the map")
    if (numbersMap.containsValue(1)) println("The value 1 is in the map") // same as previous
//sampleEnd
}
```

Two maps containing the equal pairs are equal regardless of the pair order.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
    val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)

    println("The maps are equal: ${numbersMap == anotherMap}")
//sampleEnd
}
```

[MutableMap](#) is a Map with map write operations, for example, you can add a new key-value pair or update the value associated with the given key.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    numbersMap["one"] = 11

    println(numbersMap)
//sampleEnd
}
```

The default implementation of MutableMap – LinkedHashMap – preserves the order of elements insertion when iterating the map. In turn, an alternative implementation – HashMap – says nothing about the elements order.

Constructing collections

Construct from elements

The most common way to create a collection is with the standard library functions listOf<T>(), setOf<T>(), mutableListOf<T>(), mutableSetOf<T>(). If you provide a comma-separated list of collection elements as arguments, the compiler detects the element type automatically. When creating empty collections, specify the type explicitly.

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

The same is available for maps with the functions mapOf() and mutableMapOf(). The map's keys and values are passed as Pair objects (usually created with to infix function).

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
```

Note that the to notation creates a short-living Pair object, so it's recommended that you use it only if performance isn't critical. To avoid excessive memory usage, use alternative ways. For example, you can create a mutable map and populate it using the write operations. The apply() function can help to keep the initialization fluent here.

```
val numbersMap = mutableMapOf<String, String>().apply { this["one"] = "1"; this["two"] = "2" }
```

Create with collection builder functions

Another way of creating a collection is to call a builder function – buildList(), buildSet(), or buildMap(). They create a new, mutable collection of the corresponding type, populate it using write operations, and return a read-only collection with the same elements:

```
val map = buildMap { // this is MutableMap<String, Int>, types of key and value are inferred from the `put()` calls below
    put("a", 1)
    put("b", 0)
    put("c", 4)
}

println(map) // {a=1, b=0, c=4}
```

Empty collections

There are also functions for creating collections without any elements: emptyList(), emptySet(), and emptyMap(). When creating empty collections, you should specify the type of elements that the collection will hold.

```
val empty = emptyList<String>()
```

Initializer functions for lists

For lists, there is a constructor-like function that takes the list size and the initializer function that defines the element value based on its index.

```
fun main() {
//sampleStart
    val doubled = List(3, { it * 2 })  // or MutableList if you want to change its content later
    println(doubled)
//sampleEnd
}
```

Concrete type constructors

To create a concrete type collection, such as an ArrayList or LinkedList, you can use the available constructors for these types. Similar constructors are available for implementations of Set and Map.

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))
val presizedSet = HashSet<Int>(32)
```

Copy

To create a collection with the same elements as an existing collection, you can use copying functions. Collection copying functions from the standard library create shallow copy collections with references to the same elements. Thus, a change made to a collection element reflects in all its copies.

Collection copying functions, such as toList(), toMutableList(), toSet() and others, create a snapshot of a collection at a specific moment. Their result is a new collection of the same elements. If you add or remove elements from the original collection, this won't affect the copies. Copies may be changed independently of the source as well.

```
class Person(var name: String)
fun main() {
//sampleStart
    val alice = Person("Alice")
    val sourceList = mutableListOf(alice, Person("Bob"))
    val copyList = sourceList.toList()
    sourceList.add(Person("Charles"))
    alice.name = "Alicia"
    println("First item's name is: ${sourceList[0].name} in source and ${copyList[0].name} in copy")
    println("List size is: ${sourceList.size} in source and ${copyList.size} in copy")
//sampleEnd
}
```

These functions can also be used for converting collections to other types, for example, build a set from a list or vice versa.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val copySet = sourceList.toMutableSet()
    copySet.add(3)
    copySet.add(4)
    println(copySet)
//sampleEnd
}
```

Alternatively, you can create new references to the same collection instance. New references are created when you initialize a collection variable with an existing collection. So, when the collection instance is altered through a reference, the changes are reflected in all its references.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList = sourceList
    referenceList.add(4)
    println("Source size: ${sourceList.size}")
//sampleEnd
}
```

Collection initialization can be used for restricting mutability. For example, if you create a List reference to a MutableList, the compiler will produce errors if you try to modify the collection through this reference.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList: List<Int> = sourceList
    //referenceList.add(4)            //compilation error
    sourceList.add(4)
    println(referenceList) // shows the current state of sourceList
//sampleEnd
}
```

Invoke functions on other collections

Collections can be created in result of various operations on other collections. For example, filtering a list creates a new list of elements that match the filter:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)
//sampleEnd
}
```

Mapping produces a list of a transformation results:

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
//sampleEnd
}
```

Association produces maps:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

For more information about operations on collections in Kotlin, see Collection operations overview.

Iterators

For traversing collection elements, the Kotlin standard library supports the commonly used mechanism of iterators – objects that provide access to the elements sequentially without exposing the underlying structure of the collection. Iterators are useful when you need to process all the elements of a collection one-by-one, for example, print values or make similar updates to them.

Iterators can be obtained for inheritors of the Iterable<T> interface, including Set and List, by calling the iterator() function.

Once you obtain an iterator, it points to the first element of a collection; calling the next() function returns this element and moves the iterator position to the following element if it exists.

Once the iterator passes through the last element, it can no longer be used for retrieving elements; neither can it be reset to any previous position. To iterate through the collection again, create a new iterator.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val numbersIterator = numbers.iterator()
```

```
    while (numbersIterator.hasNext()) {
        println(numbersIterator.next())
    }
//sampleEnd
}
```

Another way to go through an Iterable collection is the well-known for loop. When using for on a collection, you obtain the iterator implicitly. So, the following code is equivalent to the example above:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    for (item in numbers) {
        println(item)
    }
//sampleEnd
}
```

Finally, there is a useful forEach() function that lets you automatically iterate a collection and execute the given code for each element. So, the same example would look like this:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    numbers.forEach {
        println(it)
    }
//sampleEnd
}
```

List iterators

For lists, there is a special iterator implementation: ListIterator. It supports iterating lists in both directions: forwards and backwards.

Backward iteration is implemented by the functions hasPrevious() and previous(). Additionally, the ListIterator provides information about the element indices with the functions nextIndex() and previousIndex().

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val listIterator = numbers.listIterator()
    while (listIterator.hasNext()) listIterator.next()
    println("Iterating backwards:")
    while (listIterator.hasPrevious()) {
        print("Index: ${listIterator.previousIndex()}")
        println(", value: ${listIterator.previous()}")
    }
//sampleEnd
}
```

Having the ability to iterate in both directions, means the ListIterator can still be used after it reaches the last element.

Mutable iterators

For iterating mutable collections, there is MutableIterator that extends Iterator with the element removal function remove(). So, you can remove elements from a collection while iterating it.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val mutableIterator = numbers.iterator()

    mutableIterator.next()
    mutableIterator.remove()
```

```
    println("After removal: $numbers")
//sampleEnd
}
```

In addition to removing elements, the MutableListIterator can also insert and replace elements while iterating the list.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "four", "four")
    val mutableListIterator = numbers.listIterator()

    mutableListIterator.next()
    mutableListIterator.add("two")
    mutableListIterator.next()
    mutableListIterator.set("three")
    println(numbers)
//sampleEnd
}
```

Ranges and progressions

Kotlin lets you easily create ranges of values using the .rangeTo() and .rangeUntil() functions from the kotlin.ranges package.

To create:

- a closed-ended range, call the .rangeTo() function with the .. operator.
- an open-ended range, call the .rangeUntil() function with the ..< operator.

For example:

```
fun main() {
//sampleStart
    // Closed-ended range
    println(4 in 1..4)
    // true

    // Open-ended range
    println(4 in 1..<4)
    // false
//sampleEnd
}
```

Ranges are particularly useful for iterating over for loops:

```
fun main() {
//sampleStart
    for (i in 1..4) print(i)
    // 1234
//sampleEnd
}
```

To iterate numbers in reverse order, use the downTo function instead of ...

```
fun main() {
//sampleStart
    for (i in 4 downTo 1) print(i)
    // 4321
//sampleEnd
}
```

It is also possible to iterate over numbers with an arbitrary step (not necessarily 1). This is done via the step function.

```
fun main() {
//sampleStart
    for (i in 0..8 step 2) print(i)
    println()
```

```
    // 02468
    for (i in 0..<8 step 2) print(i)
    println()
    // 0246
    for (i in 8 downTo 0 step 2) print(i)
    // 86420
//sampleEnd
}
```

Progression

The ranges of integral types, such as Int, Long, and Char, can be treated as arithmetic progressions. In Kotlin, these progressions are defined by special types: IntProgression, LongProgression, and CharProgression.

Progressions have three essential properties: the first element, the last element, and a non-zero step. The first element is first, subsequent elements are the previous element plus a step. Iteration over a progression with a positive step is equivalent to an indexed for loop in Java/JavaScript.

```
for (int i = first; i <= last; i += step) {
  // ...
}
```

When you create a progression implicitly by iterating a range, this progression's first and last elements are the range's endpoints, and the step is 1.

```
fun main() {
//sampleStart
    for (i in 1..10) print(i)
    // 12345678910
//sampleEnd
}
```

To define a custom progression step, use the step function on a range.

```
fun main() {
//sampleStart
    for (i in 1..8 step 2) print(i)
    // 1357
//sampleEnd
}
```

The last element of the progression is calculated this way:

- For a positive step: the maximum value not greater than the end value such that (last - first) % step == 0.
- For a negative step: the minimum value not less than the end value such that (last - first) % step == 0.

Thus, the last element is not always the same as the specified end value.

```
fun main() {
//sampleStart
    for (i in 1..9 step 3) print(i) // the last element is 7
    // 147
//sampleEnd
}
```

Progressions implement Iterable<N>, where N is Int, Long, or Char respectively, so you can use them in various collection functions like map, filter, and other.

```
fun main() {
//sampleStart
    println((1..10).filter { it % 2 == 0 })
    // [2, 4, 6, 8, 10]
//sampleEnd
}
```

Sequences

Along with collections, the Kotlin standard library contains another type – sequences (Sequence<T>). Unlike collections, sequences don't contain elements, they produce them while iterating. Sequences offer the same functions as Iterable but implement another approach to multi-step collection processing.

When the processing of an Iterable includes multiple steps, they are executed eagerly: each processing step completes and returns its result – an intermediate collection. The following step executes on this collection. In turn, multi-step processing of sequences is executed lazily when possible: actual computing happens only when the result of the whole processing chain is requested.

The order of operations execution is different as well: Sequence performs all the processing steps one-by-one for every single element. In turn, Iterable completes each step for the whole collection and then proceeds to the next step.

So, the sequences let you avoid building results of intermediate steps, therefore improving the performance of the whole collection processing chain. However, the lazy nature of sequences adds some overhead which may be significant when processing smaller collections or doing simpler computations. Hence, you should consider both Sequence and Iterable and decide which one is better for your case.

Construct

From elements

To create a sequence, call the sequenceOf() function listing the elements as its arguments.

```
val numbersSequence = sequenceOf("four", "three", "two", "one")
```

From an Iterable

If you already have an Iterable object (such as a List or a Set), you can create a sequence from it by calling asSequence().

```
val numbers = listOf("one", "two", "three", "four")
val numbersSequence = numbers.asSequence()
```

From a function

One more way to create a sequence is by building it with a function that calculates its elements. To build a sequence based on a function, call generateSequence() with this function as an argument. Optionally, you can specify the first element as an explicit value or a result of a function call. The sequence generation stops when the provided function returns null. So, the sequence in the example below is infinite.

```
fun main() {
//sampleStart
    val oddNumbers = generateSequence(1) { it + 2 } // `it` is the previous element
    println(oddNumbers.take(5).toList())
    //println(oddNumbers.count())     // error: the sequence is infinite
//sampleEnd
}
```

To create a finite sequence with generateSequence(), provide a function that returns null after the last element you need.

```
fun main() {
//sampleStart
    val oddNumbersLessThan10 = generateSequence(1) { if (it < 8) it + 2 else null }
    println(oddNumbersLessThan10.count())
//sampleEnd
}
```

From chunks

Finally, there is a function that lets you produce sequence elements one by one or by chunks of arbitrary sizes – the sequence() function. This function takes a lambda expression containing calls of yield() and yieldAll() functions. They return an element to the sequence consumer and suspend the execution of sequence() until the next element is requested by the consumer. yield() takes a single element as an argument; yieldAll() can take an Iterable object, an Iterator, or another

Sequence. A Sequence argument of yieldAll() can be infinite. However, such a call must be the last: all subsequent calls will never be executed.

```
fun main() {
//sampleStart
    val oddNumbers = sequence {
        yield(1)
        yieldAll(listOf(3, 5))
        yieldAll(generateSequence(7) { it + 2 })
    }
    println(oddNumbers.take(5).toList())
//sampleEnd
}
```

Sequence operations

The sequence operations can be classified into the following groups regarding their state requirements:

- Stateless operations require no state and process each element independently, for example, map() or filter(). Stateless operations can also require a small constant amount of state to process an element, for example, take() or drop().
- Stateful operations require a significant amount of state, usually proportional to the number of elements in a sequence.

If a sequence operation returns another sequence, which is produced lazily, it's called intermediate. Otherwise, the operation is terminal. Examples of terminal operations are toList() or sum(). Sequence elements can be retrieved only with terminal operations.

Sequences can be iterated multiple times; however some sequence implementations might constrain themselves to be iterated only once. That is mentioned specifically in their documentation.

Sequence processing example

Let's take a look at the difference between Iterable and Sequence with an example.

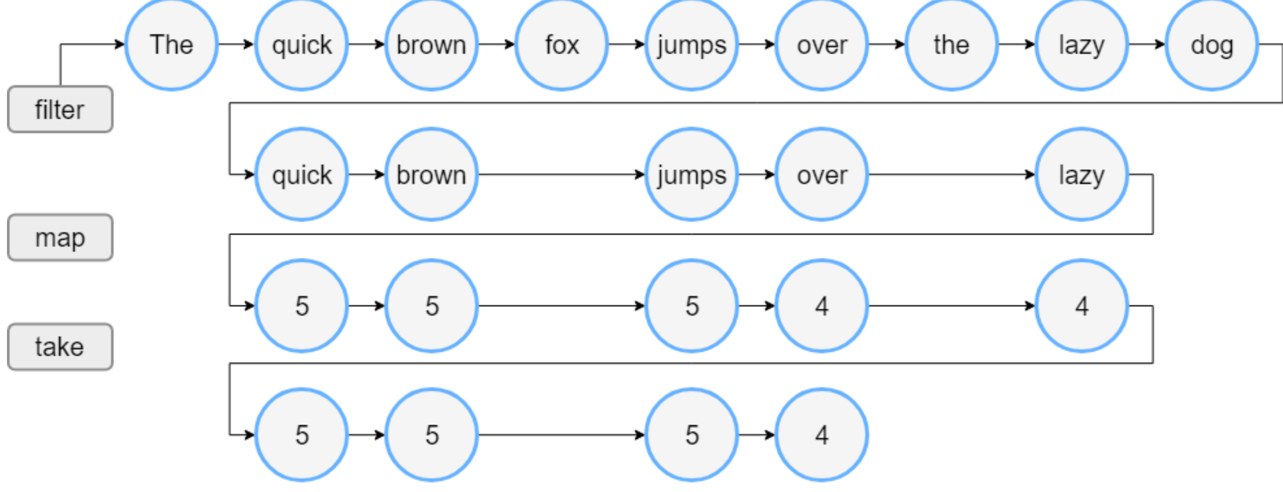
Iterable

Assume that you have a list of words. The code below filters the words longer than three characters and prints the lengths of first four such words.

```
fun main() {
//sampleStart
    val words = "The quick brown fox jumps over the lazy dog".split(" ")
    val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars:")
    println(lengthsList)
//sampleEnd
}
```

When you run this code, you'll see that the filter() and map() functions are executed in the same order as they appear in the code. First, you see filter: for all elements, then length: for the elements left after filtering, and then the output of the two last lines.

This is how the list processing goes:



List processing

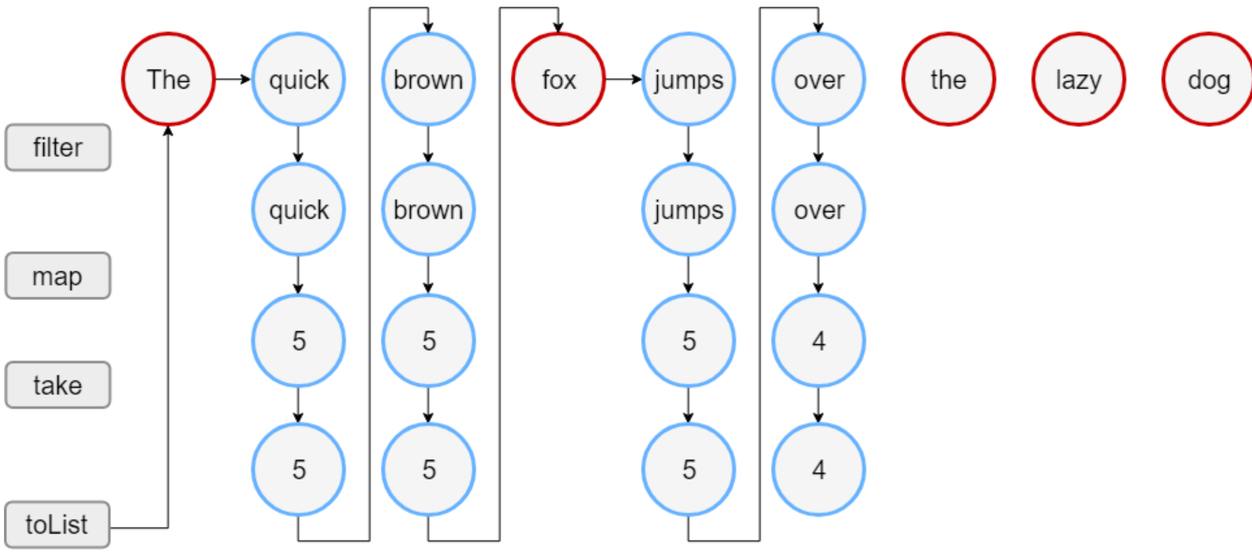
Sequence

Now let's write the same with sequences:

```
fun main() {
//sampleStart
    val words = "The quick brown fox jumps over the lazy dog".split(" ")
    //convert the List to a Sequence
    val wordsSequence = words.asSequence()

    val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars")
    // terminal operation: obtaining the result as a List
    println(lengthsSequence.toList())
//sampleEnd
}
```

The output of this code shows that the filter() and map() functions are called only when building the result list. So, you first see the line of text "Lengths of.." and then the sequence processing starts. Note that for elements left after filtering, the map executes before filtering the next element. When the result size reaches 4, the processing stops because it's the largest possible size that take(4) can return.

The sequence processing goes like this:



Sequences processing

In this example, the sequence processing takes 18 steps instead of 23 steps for doing the same with lists.

Collection operations overview

The Kotlin standard library offers a broad variety of functions for performing operations on collections. This includes simple operations, such as getting or adding elements, as well as more complex ones including search, sorting, filtering, transformations, and so on.

Extension and member functions

Collection operations are declared in the standard library in two ways: member functions of collection interfaces and extension functions.

Member functions define operations that are essential for a collection type. For example, Collection contains the function isEmpty() for checking its emptiness; List contains get() for index access to elements, and so on.

When you create your own implementations of collection interfaces, you must implement their member functions. To make the creation of new implementations easier, use the skeletal implementations of collection interfaces from the standard library: AbstractCollection, AbstractList, AbstractSet, AbstractMap, and their mutable counterparts.

Other collection operations are declared as extension functions. These are filtering, transformation, ordering, and other collection processing functions.

Common operations

Common operations are available for both read-only and mutable collections. Common operations fall into these groups:

- Transformations
- Filtering
- plus and minus operators
- Grouping
- Retrieving collection parts
- Retrieving single elements
- Ordering
- Aggregate operations

Operations described on these pages return their results without affecting the original collection. For example, a filtering operation produces a new collection that contains all the elements matching the filtering predicate. Results of such operations should be either stored in variables, or used in some other way, for example, passed in other functions.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    numbers.filter { it.length > 3 }  // nothing happens with `numbers`, result is lost
    println("numbers are still $numbers")
    val longerThan3 = numbers.filter { it.length > 3 } // result is stored in `longerThan3`
    println("numbers longer than 3 chars are $longerThan3")
//sampleEnd
}
```

For certain collection operations, there is an option to specify the destination object. Destination is a mutable collection to which the function appends its resulting items instead of returning them in a new object. For performing operations with destinations, there are separate functions with the To postfix in their names, for example, filterTo() instead of filter() or associateTo() instead of associate(). These functions take the destination collection as an additional parameter.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val filterResults = mutableListOf<String>()  //destination object
    numbers.filterTo(filterResults) { it.length > 3 }
    numbers.filterIndexedTo(filterResults) { index, _ -> index == 0 }
    println(filterResults) // contains results of both operations
//sampleEnd
}
```

For convenience, these functions return the destination collection back, so you can create it right in the corresponding argument of the function call:

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
//sampleStart
    // filter numbers right into a new hash set,
    // thus eliminating duplicates in the result
    val result = numbers.mapTo(HashSet()) { it.length }
    println("distinct item lengths are $result")
//sampleEnd
}
```

Functions with destination are available for filtering, association, grouping, flattening, and other operations. For the complete list of destination operations see the Kotlin collections reference.

Write operations

For mutable collections, there are also write operations that change the collection state. Such operations include adding, removing, and updating elements. Write operations are listed in the Write operations and corresponding sections of List-specific operations and Map specific operations.

For certain operations, there are pairs of functions for performing the same operation: one applies the operation in-place and the other returns the result as a separate collection. For example, sort() sorts a mutable collection in-place, so its state changes; sorted() creates a new collection that contains the same elements in the sorted order.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val sortedNumbers = numbers.sorted()
    println(numbers == sortedNumbers)  // false
    numbers.sort()
    println(numbers == sortedNumbers)  // true
//sampleEnd
}
```

Collection transformation operations

The Kotlin standard library provides a set of extension functions for collection transformations. These functions build new collections from existing ones based on the transformation rules provided. In this page, we'll give an overview of the available collection transformation functions.

Map

The mapping transformation creates a collection from the results of a function on the elements of another collection. The basic mapping function is map(). It applies the given lambda function to each subsequent element and returns the list of the lambda results. The order of results is the same as the original order of elements. To apply a transformation that additionally uses the element index as an argument, use mapIndexed().

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
//sampleEnd
}
```

If the transformation produces null on certain elements, you can filter out the nulls from the result collection by calling the mapNotNull() function instead of map(), or mapIndexedNotNull() instead of mapIndexed().

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.mapNotNull { if ( it == 2) null else it * 3 })
    println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value * idx })
//sampleEnd
}
```

When transforming maps, you have two options: transform keys leaving values unchanged and vice versa. To apply a given transformation to keys, use mapKeys(); in turn, mapValues() transforms values. Both functions use the transformations that take a map entry as an argument, so you can operate both its key and value.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    println(numbersMap.mapKeys { it.key.uppercase() })
    println(numbersMap.mapValues { it.value + it.key.length })
//sampleEnd
}
```

Zip

Zipping transformation is building pairs from elements with the same positions in both collections. In the Kotlin standard library, this is done by the zip() extension function.

When called on a collection or an array with another collection (or array) as an argument, zip() returns the List of Pair objects. The elements of the receiver collection are the first elements in these pairs.

If the collections have different sizes, the result of the zip() is the smaller size; the last elements of the larger collection are not included in the result.

zip() can also be called in the infix form a zip b.

```
fun main() {
//sampleStart
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")
    println(colors zip animals)

    val twoAnimals = listOf("fox", "bear")
    println(colors.zip(twoAnimals))
//sampleEnd
```

```
}
```

You can also call zip() with a transformation function that takes two parameters: the receiver element and the argument element. In this case, the result List contains the return values of the transformation function called on pairs of the receiver and the argument elements with the same positions.

```
fun main() {
//sampleStart
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal -> "The ${animal.replaceFirstChar { it.uppercase() }} is $color"})
//sampleEnd
}
```

When you have a List of Pairs, you can do the reverse transformation – unzipping – that builds two lists from these pairs:

- The first list contains the first elements of each Pair in the original list.
- The second list contains the second elements.

To unzip a list of pairs, call unzip().

```
fun main() {
//sampleStart
    val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
    println(numberPairs.unzip())
//sampleEnd
}
```

Associate

Association transformations allow building maps from the collection elements and certain values associated with them. In different association types, the elements can be either keys or values in the association map.

The basic association function associateWith() creates a Map in which the elements of the original collection are keys, and values are produced from them by the given transformation function. If two elements are equal, only the last one remains in the map.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

For building maps with collection elements as values, there is the function associateBy(). It takes a function that returns a key based on an element's value. If two elements' keys are equal, only the last one remains in the map.

associateBy() can also be called with a value transformation function.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.associateBy { it.first().uppercaseChar() })
    println(numbers.associateBy(keySelector = { it.first().uppercaseChar() }, valueTransform = { it.length }))
//sampleEnd
}
```

Another way to build maps in which both keys and values are somehow produced from collection elements is the function associate(). It takes a lambda function that returns a Pair: the key and the value of the corresponding map entry.

Note that associate() produces short-living Pair objects which may affect the performance. Thus, associate() should be used when the performance isn't critical or it's more preferable than other options.

An example of the latter is when a key and the corresponding value are produced from an element together.

```
fun main() {
data class FullName (val firstName: String, val lastName: String)

fun parseFullName(fullName: String): FullName {
    val nameParts = fullName.split(" ")
    if (nameParts.size == 2) {
        return FullName(nameParts[0], nameParts[1])
    } else throw Exception("Wrong name format")
}

//sampleStart
    val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")
    println(names.associate { name -> parseFullName(name).let { it.lastName to it.firstName } })
//sampleEnd
}
```

Here we call a transform function on an element first, and then build a pair from the properties of that function's result.

Flatten

If you operate nested collections, you may find the standard library functions that provide flat access to nested collection elements useful.

The first function is flatten(). You can call it on a collection of collections, for example, a List of Sets. The function returns a single List of all the elements of the nested collections.

```
fun main() {
//sampleStart
    val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1, 2))
    println(numberSets.flatten())
//sampleEnd
}
```

Another function – flatMap() provides a flexible way to process nested collections. It takes a function that maps a collection element to another collection. As a result, flatMap() returns a single list of its return values on all the elements. So, flatMap() behaves as a subsequent call of map() (with a collection as a mapping result) and flatten().

```
data class StringContainer(val values: List<String>)

fun main() {
//sampleStart
    val containers = listOf(
        StringContainer(listOf("one", "two", "three")),
        StringContainer(listOf("four", "five", "six")),
        StringContainer(listOf("seven", "eight"))
    )
    println(containers.flatMap { it.values })
//sampleEnd
}
```

String representation

If you need to retrieve the collection content in a readable format, use functions that transform the collections to strings: joinToString() and joinTo().

joinToString() builds a single String from the collection elements based on the provided arguments. joinTo() does the same but appends the result to the given Appendable object.

When called with the default arguments, the functions return the result similar to calling toString() on the collection: a String of elements' string representations separated by commas with spaces.

```
fun main() {
//sampleStart
```

```
    val numbers = listOf("one", "two", "three", "four")

    println(numbers)
    println(numbers.joinToString())

    val listString = StringBuffer("The list of numbers: ")
    numbers.joinTo(listString)
    println(listString)
//sampleEnd
}
```

To build a custom string representation, you can specify its parameters in function arguments separator, prefix, and postfix. The resulting string will start with the prefix and end with the postfix. The separator will come after each element except the last.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": end"))
//sampleEnd
}
```

For bigger collections, you may want to specify the limit – a number of elements that will be included into result. If the collection size exceeds the limit, all the other elements will be replaced with a single value of the truncated argument.

```
fun main() {
//sampleStart
    val numbers = (1..100).toList()
    println(numbers.joinToString(limit = 10, truncated = "<...>"))
//sampleEnd
}
```

Finally, to customize the representation of elements themselves, provide the transform function.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString { "Element: ${it.uppercase()}"})
//sampleEnd
}
```

Filtering collections

Filtering is one of the most popular tasks in collection processing. In Kotlin, filtering conditions are defined by predicates – lambda functions that take a collection element and return a boolean value: true means that the given element matches the predicate, false means the opposite.

The standard library contains a group of extension functions that let you filter collections in a single call. These functions leave the original collection unchanged, so they are available for both mutable and read-only collections. To operate the filtering result, you should assign it to a variable or chain the functions after filtering.

Filter by predicate

The basic filtering function is filter(). When called with a predicate, filter() returns the collection elements that match it. For both List and Set, the resulting collection is a List, for Map it's a Map as well.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)

    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
    println(filteredMap)
//sampleEnd
}
```

The predicates in filter() can only check the values of the elements. If you want to use element positions in the filter, use filterIndexed(). It takes a predicate with two arguments: the index and the value of an element.

To filter collections by negative conditions, use filterNot(). It returns a list of elements for which the predicate yields false.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5)  }
    val filteredNot = numbers.filterNot { it.length <= 3 }

    println(filteredIdx)
    println(filteredNot)
//sampleEnd
}
```

There are also functions that narrow the element type by filtering elements of a given type:

- filterIsInstance() returns collection elements of a given type. Being called on a List<Any>, filterIsInstance<T>() returns a List<T>, thus allowing you to call functions of the T type on its items.

```
fun main() {
//sampleStart
    val numbers = listOf(null, 1, "two", 3.0, "four")
    println("All String elements in upper case:")
    numbers.filterIsInstance<String>().forEach {
        println(it.uppercase())
    }
//sampleEnd
}
```

- filterNotNull() returns all non-null elements. Being called on a List<T?>, filterNotNull() returns a List<T: Any>, thus allowing you to treat the elements as non-null objects.

```
fun main() {
//sampleStart
    val numbers = listOf(null, "one", "two", null)
    numbers.filterNotNull().forEach {
        println(it.length)   // length is unavailable for nullable Strings
    }
//sampleEnd
}
```

Partition

Another filtering function – partition() – filters a collection by a predicate and keeps the elements that don't match it in a separate list. So, you have a Pair of Lists as a return value: the first list containing elements that match the predicate and the second one containing everything else from the original collection.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val (match, rest) = numbers.partition { it.length > 3 }

    println(match)
    println(rest)
//sampleEnd
}
```

Test predicates

Finally, there are functions that simply test a predicate against collection elements:

- any() returns true if at least one element matches the given predicate.
- none() returns true if none of the elements match the given predicate.

- all() returns true if all elements match the given predicate. Note that all() returns true when called with any valid predicate on an empty collection. Such behavior is known in logic as vacuous truth.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.any { it.endsWith("e") })
    println(numbers.none { it.endsWith("a") })
    println(numbers.all { it.endsWith("e") })

    println(emptyList<Int>().all { it > 5 })   // vacuous truth
//sampleEnd
}
```

any() and none() can also be used without a predicate: in this case they just check the collection emptiness. any() returns true if there are elements and false if there aren't; none() does the opposite.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val empty = emptyList<String>()

    println(numbers.any())
    println(empty.any())

    println(numbers.none())
    println(empty.none())
//sampleEnd
}
```

Plus and minus operators

In Kotlin, plus (+) and minus (-) operators are defined for collections. They take a collection as the first operand; the second operand can be either an element or another collection. The return value is a new read-only collection:

- The result of plus contains the elements from the original collection and from the second operand.
- The result of minus contains the elements of the original collection except the elements from the second operand. If it's an element, minus removes its first occurrence; if it's a collection, all occurrences of its elements are removed.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val plusList = numbers + "five"
    val minusList = numbers - listOf("three", "four")
    println(plusList)
    println(minusList)
//sampleEnd
}
```

For the details on plus and minus operators for maps, see Map specific operations. The augmented assignment operators plusAssign (+=) and minusAssign (-=) are also defined for collections. However, for read-only collections, they actually use the plus or minus operators and try to assign the result to the same variable. Thus, they are available only on var read-only collections. For mutable collections, they modify the collection if it's a val. For more details see Collection write operations.

Grouping

The Kotlin standard library provides extension functions for grouping collection elements. The basic function groupBy() takes a lambda function and returns a Map. In this map, each key is the lambda result and the corresponding value is the List of elements on which this result is returned. This function can be used, for example, to group a list of Strings by their first letter.

You can also call groupBy() with a second lambda argument – a value transformation function. In the result map of groupBy() with two lambdas, the keys produced by keySelector function are mapped to the results of the value transformation function instead of the original elements.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")

    println(numbers.groupBy { it.first().uppercase() })
    println(numbers.groupBy(keySelector = { it.first() }, valueTransform = { it.uppercase() }))
//sampleEnd
}
```

If you want to group elements and then apply an operation to all groups at one time, use the function groupingBy(). It returns an instance of the Grouping type. The Grouping instance lets you apply operations to all groups in a lazy manner: the groups are actually built right before the operation execution.

Namely, Grouping supports the following operations:

- eachCount() counts the elements in each group.
- fold() and reduce() perform fold and reduce operations on each group as a separate collection and return the results.
- aggregate() applies a given operation subsequently to all the elements in each group and returns the result. This is the generic way to perform any operations on a Grouping. Use it to implement custom operations when fold or reduce are not enough.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.groupingBy { it.first() }.eachCount())
//sampleEnd
}
```

Retrieve collection parts

The Kotlin standard library contains extension functions for retrieving parts of a collection. These functions provide a variety of ways to select elements for the result collection: listing their positions explicitly, specifying the result size, and others.

Slice

slice() returns a list of the collection elements with given indices. The indices may be passed either as a range or as a collection of integer values.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.slice(1..3))
    println(numbers.slice(0..4 step 2))
    println(numbers.slice(setOf(3, 5, 0)))
//sampleEnd
}
```

Take and drop

To get the specified number of elements starting from the first, use the take() function. For getting the last elements, use takeLast(). When called with a number larger than the collection size, both functions return the whole collection.

To take all the elements except a given number of first or last elements, call the drop() and dropLast() functions respectively.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.take(3))
    println(numbers.takeLast(3))
    println(numbers.drop(1))
    println(numbers.dropLast(5))
//sampleEnd
}
```

You can also use predicates to define the number of elements for taking or dropping. There are four functions similar to the ones described above:

- takeWhile() is take() with a predicate: it takes the elements up to but excluding the first one not matching the predicate. If the first collection element doesn't match the predicate, the result is empty.
- takeLastWhile() is similar to takeLast(): it takes the range of elements matching the predicate from the end of the collection. The first element of the range is the element next to the last element not matching the predicate. If the last collection element doesn't match the predicate, the result is empty;
- dropWhile() is the opposite to takeWhile() with the same predicate: it returns the elements from the first one not matching the predicate to the end.
- dropLastWhile() is the opposite to takeLastWhile() with the same predicate: it returns the elements from the beginning to the last one not matching the predicate.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.takeWhile { !it.startsWith('f') })
    println(numbers.takeLastWhile { it != "three" })
    println(numbers.dropWhile { it.length == 3 })
    println(numbers.dropLastWhile { it.contains('i') })
//sampleEnd
}
```

Chunked

To break a collection into parts of a given size, use the chunked() function. chunked() takes a single argument – the size of the chunk – and returns a List of Lists of the given size. The first chunk starts from the first element and contains the size elements, the second chunk holds the next size elements, and so on. The last chunk may have a smaller size.

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.chunked(3))
//sampleEnd
}
```

You can also apply a transformation for the returned chunks right away. To do this, provide the transformation as a lambda function when calling chunked(). The lambda argument is a chunk of the collection. When chunked() is called with a transformation, the chunks are short-living Lists that should be consumed right in that lambda.

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.chunked(3) { it.sum() })  // `it` is a chunk of the original collection
//sampleEnd
}
```

Windowed

You can retrieve all possible ranges of the collection elements of a given size. The function for getting them is called windowed(): it returns a list of element ranges that you would see if you were looking at the collection through a sliding window of the given size. Unlike chunked(), windowed() returns element ranges (windows) starting from each collection element. All the windows are returned as elements of a single List.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.windowed(3))
//sampleEnd
}
```

windowed() provides more flexibility with optional parameters:

- step defines a distance between first elements of two adjacent windows. By default the value is 1, so the result contains windows starting from all elements. If you increase the step to 2, you will receive only windows starting from odd elements: first, third, and so on.

- partialWindows includes windows of smaller sizes that start from the elements at the end of the collection. For example, if you request windows of three elements, you can't build them for the last two elements. Enabling partialWindows in this case includes two more lists of sizes 2 and 1.

Finally, you can apply a transformation to the returned ranges right away. To do this, provide the transformation as a lambda function when calling windowed().

```
fun main() {
//sampleStart
    val numbers = (1..10).toList()
    println(numbers.windowed(3, step = 2, partialWindows = true))
    println(numbers.windowed(3) { it.sum() })
//sampleEnd
}
```

To build two-element windows, there is a separate function - zipWithNext(). It creates pairs of adjacent elements of the receiver collection. Note that zipWithNext() doesn't break the collection into pairs; it creates a Pair for each element except the last one, so its result on [1, 2, 3, 4] is [[1, 2], [2, 3], [3, 4]], not [[1, 2], [3, 4]]. zipWithNext() can be called with a transformation function as well; it should take two elements of the receiver collection as arguments.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.zipWithNext())
    println(numbers.zipWithNext() { s1, s2 -> s1.length > s2.length})
//sampleEnd
}
```

Retrieve single elements

Kotlin collections provide a set of functions for retrieving single elements from collections. Functions described on this page apply to both lists and sets.

As the definition of list says, a list is an ordered collection. Hence, every element of a list has its position that you can use for referring. In addition to functions described on this page, lists offer a wider set of ways to retrieve and search for elements by indices. For more details, see List-specific operations.

In turn, set is not an ordered collection by definition. However, the Kotlin Set stores elements in certain orders. These can be the order of insertion (in LinkedHashSet), natural sorting order (in SortedSet), or another order. The order of a set of elements can also be unknown. In such cases, the elements are still ordered somehow, so the functions that rely on the element positions still return their results. However, such results are unpredictable to the caller unless they know the specific implementation of Set used.

Retrieve by position

For retrieving an element at a specific position, there is the function elementAt(). Call it with the integer number as an argument, and you'll receive the collection element at the given position. The first element has the position 0, and the last one is (size - 1).

elementAt() is useful for collections that do not provide indexed access, or are not statically known to provide one. In case of List, it's more idiomatic to use indexed access operator (get() or []).

```
fun main() {
//sampleStart
    val numbers = linkedSetOf("one", "two", "three", "four", "five")
    println(numbers.elementAt(3))

    val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
    println(numbersSortedSet.elementAt(0)) // elements are stored in the ascending order
//sampleEnd
}
```

There are also useful aliases for retrieving the first and the last element of the collection: first() and last().

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.first())
```

```
    println(numbers.last())
//sampleEnd
}
```

To avoid exceptions when retrieving element with non-existing positions, use safe variations of elementAt():

- elementAtOrNull() returns null when the specified position is out of the collection bounds.
- elementAtOrElse() additionally takes a lambda function that maps an Int argument to an instance of the collection element type. When called with an out-of-bounds position, the elementAtOrElse() returns the result of the lambda on the given value.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.elementAtOrNull(5))
    println(numbers.elementAtOrElse(5) { index -> "The value for index $index is undefined"})
//sampleEnd
}
```

Retrieve by condition

Functions first() and last() also let you search a collection for elements matching a given predicate. When you call first() with a predicate that tests a collection element, you'll receive the first element on which the predicate yields true. In turn, last() with a predicate returns the last element matching it.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.first { it.length > 3 })
    println(numbers.last { it.startsWith("f") })
//sampleEnd
}
```

If no elements match the predicate, both functions throw exceptions. To avoid them, use firstOrNull() and lastOrNull() instead: they return null if no matching elements are found.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.firstOrNull { it.length > 6 })
//sampleEnd
}
```

Use the aliases if their names suit your situation better:

- find() instead of firstOrNull()
- findLast() instead of lastOrNull()

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.find { it % 2 == 0 })
    println(numbers.findLast { it % 2 == 0 })
//sampleEnd
}
```

Retrieve with selector

If you need to map the collection before retrieving the element, there is a function firstNotNullOf(). It combines 2 actions:

- Maps the collection with the selector function
- Returns the first non-null value in the result

firstNotNullOf() throws the NoSuchElementException if the resulting collection doesn't have a non-null element. Use the counterpart firstNotNullOfOrNull() to return null in this case.

```
fun main() {
//sampleStart
    val list = listOf<Any>(0, "true", false)
    // Converts each element to string and returns the first one that has required length
    val longEnough = list.firstNotNullOf { item -> item.toString().takeIf { it.length >= 4 } }
    println(longEnough)
//sampleEnd
}
```

Random element

If you need to retrieve an arbitrary element of a collection, call the random() function. You can call it without arguments or with a Random object as a source of the randomness.

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.random())
//sampleEnd
}
```

On empty collections, random() throws an exception. To receive null instead, use randomOrNull()

Check element existence

To check the presence of an element in a collection, use the contains() function. It returns true if there is a collection element that equals() the function argument. You can call contains() in the operator form with the in keyword.

To check the presence of multiple instances together at once, call containsAll() with a collection of these instances as an argument.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.contains("four"))
    println("zero" in numbers)

    println(numbers.containsAll(listOf("four", "two")))
    println(numbers.containsAll(listOf("one", "zero")))
//sampleEnd
}
```

Additionally, you can check if the collection contains any elements by calling isEmpty() or isNotEmpty().

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.isEmpty())
    println(numbers.isNotEmpty())

    val empty = emptyList<String>()
    println(empty.isEmpty())
    println(empty.isNotEmpty())
//sampleEnd
}
```

Ordering

The order of elements is an important aspect of certain collection types. For example, two lists of the same elements are not equal if their elements are ordered

differently.

In Kotlin, the orders of objects can be defined in several ways.

First, there is natural order. It is defined for implementations of the Comparable interface. Natural order is used for sorting them when no other order is specified.

Most built-in types are comparable:

- Numeric types use the traditional numerical order: 1 is greater than 0; -3.4f is greater than -5f, and so on.
- Char and String use the lexicographical order: b is greater than a; world is greater than hello.

To define a natural order for a user-defined type, make the type an implementer of Comparable. This requires implementing the compareTo() function. compareTo() must take another object of the same type as an argument and return an integer value showing which object is greater:

- Positive values show that the receiver object is greater.
- Negative values show that it's less than the argument.
- Zero shows that the objects are equal.

Below is a class for ordering versions that consist of the major and the minor part.

```
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int = when {
        this.major != other.major -> this.major compareTo other.major // compareTo() in the infix form
        this.minor != other.minor -> this.minor compareTo other.minor
        else -> 0
    }
}

fun main() {
    println(Version(1, 2) > Version(1, 3))
    println(Version(2, 0) > Version(1, 5))
}
```

Custom orders let you sort instances of any type in a way you like. Particularly, you can define an order for non-comparable objects or define an order other than natural for a comparable type. To define a custom order for a type, create a Comparator for it. Comparator contains the compare() function: it takes two instances of a class and returns the integer result of the comparison between them. The result is interpreted in the same way as the result of a compareTo() as is described above.

```
fun main() {
//sampleStart
    val lengthComparator = Comparator { str1: String, str2: String -> str1.length - str2.length }
    println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))
//sampleEnd
}
```

Having the lengthComparator, you are able to arrange strings by their length instead of the default lexicographical order.

A shorter way to define a Comparator is the compareBy() function from the standard library. compareBy() takes a lambda function that produces a Comparable value from an instance and defines the custom order as the natural order of the produced values.

With compareBy(), the length comparator from the example above looks like this:

```
fun main() {
//sampleStart
    println(listOf("aaa", "bb", "c").sortedWith(compareBy { it.length }))
//sampleEnd
}
```

The Kotlin collections package provides functions for sorting collections in natural, custom, and even random orders. On this page, we'll describe sorting functions that apply to read-only collections. These functions return their result as a new collection containing the elements of the original collection in the requested order. To learn about functions for sorting mutable collections in place, see the List-specific operations.

Natural order

The basic functions sorted() and sortedDescending() return elements of a collection sorted into ascending and descending sequence according to their natural order. These functions apply to collections of Comparable elements.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println("Sorted ascending: ${numbers.sorted()}")
    println("Sorted descending: ${numbers.sortedDescending()}")
//sampleEnd
}
```

Custom orders

For sorting in custom orders or sorting non-comparable objects, there are the functions sortedBy() and sortedByDescending(). They take a selector function that maps collection elements to Comparable values and sort the collection in natural order of that values.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val sortedNumbers = numbers.sortedBy { it.length }
    println("Sorted by length ascending: $sortedNumbers")
    val sortedByLast = numbers.sortedByDescending { it.last() }
    println("Sorted by the last letter descending: $sortedByLast")
//sampleEnd
}
```

To define a custom order for the collection sorting, you can provide your own Comparator. To do this, call the sortedWith() function passing in your Comparator. With this function, sorting strings by their length looks like this:

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println("Sorted by length ascending: ${numbers.sortedWith(compareBy { it.length })}")
//sampleEnd
}
```

Reverse order

You can retrieve the collection in the reversed order using the reversed() function.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.reversed())
//sampleEnd
}
```

reversed() returns a new collection with the copies of the elements. So, if you change the original collection later, this won't affect the previously obtained results of reversed().

Another reversing function - asReversed()

- returns a reversed view of the same collection instance, so it may be more lightweight and preferable than reversed() if the original list is not going to change.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
//sampleEnd
}
```

If the original list is mutable, all its changes reflect in its reversed views and vice versa.

```kotlin
fun main() {
```

```kotlin
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
    numbers.add("five")
    println(reversedNumbers)
//sampleEnd
}
```

However, if the mutability of the list is unknown or the source is not a list at all, reversed() is more preferable since its result is a copy that won't change in the future.

Random order

Finally, there is a function that returns a new List containing the collection elements in a random order - shuffled(). You can call it without arguments or with a Random object.

```kotlin
fun main() {
//sampleStart
     val numbers = listOf("one", "two", "three", "four")
     println(numbers.shuffled())
//sampleEnd
}
```

Aggregate operations

Kotlin collections contain functions for commonly used aggregate operations – operations that return a single value based on the collection content. Most of them are well known and work the same way as they do in other languages:

- minOrNull() and maxOrNull() return the smallest and the largest element respectively. On empty collections, they return null.
- average() returns the average value of elements in the collection of numbers.
- sum() returns the sum of elements in the collection of numbers.
- count() returns the number of elements in a collection.

```kotlin
fun main() {
    val numbers = listOf(6, 42, 10, 4)

    println("Count: ${numbers.count()}")
    println("Max: ${numbers.maxOrNull()}")
    println("Min: ${numbers.minOrNull()}")
    println("Average: ${numbers.average()}")
    println("Sum: ${numbers.sum()}")
}
```

There are also functions for retrieving the smallest and the largest elements by certain selector function or custom Comparator:

- maxByOrNull() and minByOrNull() take a selector function and return the element for which it returns the largest or the smallest value.
- maxWithOrNull() and minWithOrNull() take a Comparator object and return the largest or smallest element according to that Comparator.
- maxOfOrNull() and minOfOrNull() take a selector function and return the largest or the smallest return value of the selector itself.
- maxOfWithOrNull() and minOfWithOrNull() take a Comparator object and return the largest or smallest selector return value according to that Comparator.

These functions return null on empty collections. There are also alternatives – maxOf, minOf, maxOfWith, and minOfWith – which do the same as their counterparts but throw a NoSuchElementException on empty collections.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf(5, 42, 10, 4)
    val min3Remainder = numbers.minByOrNull { it % 3 }
    println(min3Remainder)

    val strings = listOf("one", "two", "three", "four")
```

```
    val longestString = strings.maxWithOrNull(compareBy { it.length })
    println(longestString)
//sampleEnd
}
```

Besides regular sum(), there is an advanced summation function sumOf() that takes a selector function and returns the sum of its application to all collection elements. Selector can return different numeric types: Int, Long, Double, UInt, and ULong (also BigInteger and BigDecimal on the JVM).

```
fun main() {
//sampleStart
    val numbers = listOf(5, 42, 10, 4)
    println(numbers.sumOf { it * 2 })
    println(numbers.sumOf { it.toDouble() / 2 })
//sampleEnd
}
```

Fold and reduce

For more specific cases, there are the functions reduce() and fold() that apply the provided operation to the collection elements sequentially and return the accumulated result. The operation takes two arguments: the previously accumulated value and the collection element.

The difference between the two functions is that fold() takes an initial value and uses it as the accumulated value on the first step, whereas the first step of reduce() uses the first and the second elements as operation arguments on the first step.

```
fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)

    val simpleSum = numbers.reduce { sum, element -> sum + element }
    println(simpleSum)
    val sumDoubled = numbers.fold(0) { sum, element -> sum + element * 2 }
    println(sumDoubled)

    //incorrect: the first element isn't doubled in the result
    //val sumDoubledReduce = numbers.reduce { sum, element -> sum + element * 2 }
    //println(sumDoubledReduce)
//sampleEnd
}
```

The example above shows the difference: fold() is used for calculating the sum of doubled elements. If you pass the same function to reduce(), it will return another result because it uses the list's first and second elements as arguments on the first step, so the first element won't be doubled.

To apply a function to elements in the reverse order, use functions reduceRight() and foldRight(). They work in a way similar to fold() and reduce() but start from the last element and then continue to previous. Note that when folding or reducing right, the operation arguments change their order: first goes the element, and then the accumulated value.

```
fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)
    val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum + element * 2 }
    println(sumDoubledRight)
//sampleEnd
}
```

You can also apply operations that take element indices as parameters. For this purpose, use functions reduceIndexed() and foldIndexed() passing element index as the first argument of the operation.

Finally, there are functions that apply such operations to collection elements from right to left - reduceRightIndexed() and foldRightIndexed().

```
fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)
    val sumEven = numbers.foldIndexed(0) { idx, sum, element -> if (idx % 2 == 0) sum + element else sum }
    println(sumEven)

    val sumEvenRight = numbers.foldRightIndexed(0) { idx, element, sum -> if (idx % 2 == 0) sum + element else sum }
```

```
    println(sumEvenRight)
//sampleEnd
}
```

All reduce operations throw an exception on empty collections. To receive null instead, use their *OrNull() counterparts:

- reduceOrNull()
- reduceRightOrNull()
- reduceIndexedOrNull()
- reduceRightIndexedOrNull()

For cases where you want to save intermediate accumulator values, there are functions runningFold() (or its synonym scan()) and runningReduce().

```
fun main() {
//sampleStart
    val numbers = listOf(0, 1, 2, 3, 4, 5)
    val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }
    val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }
//sampleEnd
    val transform = { index: Int, element: Int -> "N = ${index + 1}: $element" }
    println(runningReduceSum.mapIndexed(transform).joinToString("\n", "Sum of first N elements with runningReduce:\n"))
    println(runningFoldSum.mapIndexed(transform).joinToString("\n", "Sum of first N elements with runningFold:\n"))
}
```

If you need an index in the operation parameter, use runningFoldIndexed() or runningReduceIndexed().

Collection write operations

Mutable collections support operations for changing the collection contents, for example, adding or removing elements. On this page, we'll describe write operations available for all implementations of MutableCollection. For more specific operations available for List and Map, see List-specific Operations and Map Specific Operations respectively.

Adding elements

To add a single element to a list or a set, use the add() function. The specified object is appended to the end of the collection.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    println(numbers)
//sampleEnd
}
```

addAll() adds every element of the argument object to a list or a set. The argument can be an Iterable, a Sequence, or an Array. The types of the receiver and the argument may be different, for example, you can add all items from a Set to a List.

When called on lists, addAll() adds new elements in the same order as they go in the argument. You can also call addAll() specifying an element position as the first argument. The first element of the argument collection will be inserted at this position. Other elements of the argument collection will follow it, shifting the receiver elements to the end.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 5, 6)
    numbers.addAll(arrayOf(7, 8))
    println(numbers)
    numbers.addAll(2, setOf(3, 4))
    println(numbers)
//sampleEnd
}
```

You can also add elements using the in-place version of the plus operator - plusAssign (+=) When applied to a mutable collection, += appends the second operand (an element or another collection) to the end of the collection.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two")
    numbers += "three"
    println(numbers)
    numbers += listOf("four", "five")
    println(numbers)
//sampleEnd
}
```

Removing elements

To remove an element from a mutable collection, use the remove() function. remove() accepts the element value and removes one occurrence of this value.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.remove(3)                    // removes the first `3`
    println(numbers)
    numbers.remove(5)                    // removes nothing
    println(numbers)
//sampleEnd
}
```

For removing multiple elements at once, there are the following functions :

- removeAll() removes all elements that are present in the argument collection. Alternatively, you can call it with a predicate as an argument; in this case the function removes all elements for which the predicate yields true.
- retainAll() is the opposite of removeAll(): it removes all elements except the ones from the argument collection. When used with a predicate, it leaves only elements that match it.
- clear() removes all elements from a list and leaves it empty.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers)
    numbers.retainAll { it >= 3 }
    println(numbers)
    numbers.clear()
    println(numbers)

    val numbersSet = mutableSetOf("one", "two", "three", "four")
    numbersSet.removeAll(setOf("one", "two"))
    println(numbersSet)
//sampleEnd
}
```

Another way to remove elements from a collection is with the minusAssign (-=) operator – the in-place version of minus. The second argument can be a single instance of the element type or another collection. With a single element on the right-hand side, -= removes the first occurrence of it. In turn, if it's a collection, all occurrences of its elements are removed. For example, if a list contains duplicate elements, they are removed at once. The second operand can contain elements that are not present in the collection. Such elements don't affect the operation execution.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "three", "four")
    numbers -= "three"
    println(numbers)
    numbers -= listOf("four", "five")
    //numbers -= listOf("four")    // does the same as above
    println(numbers)
//sampleEnd
```

```
}
```

Updating elements

Lists and maps also provide operations for updating elements. They are described in List-specific Operations and Map Specific Operations. For sets, updating doesn't make sense since it's actually removing an element and adding another one.

List-specific operations

List is the most popular type of built-in collection in Kotlin. Index access to the elements of lists provides a powerful set of operations for lists.

Retrieve elements by index

Lists support all common operations for element retrieval: elementAt(), first(), last(), and others listed in Retrieve single elements. What is specific for lists is index access to the elements, so the simplest way to read an element is retrieving it by index. That is done with the get() function with the index passed in the argument or the shorthand [index] syntax.

If the list size is less than the specified index, an exception is thrown. There are two other functions that help you avoid such exceptions:

- getOrElse() lets you provide the function for calculating the default value to return if the index isn't present in the collection.
- getOrNull() returns null as the default value.

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.get(0))
    println(numbers[0])
    //numbers.get(5)                         // exception!
    println(numbers.getOrNull(5))             // null
    println(numbers.getOrElse(5, {it}))        // 5
//sampleEnd
}
```

Retrieve list parts

In addition to common operations for Retrieving Collection Parts, lists provide the subList() function that returns a view of the specified elements range as a list. Thus, if an element of the original collection changes, it also changes in the previously created sublists and vice versa.

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.subList(3, 6))
//sampleEnd
}
```

Find element positions

Linear search

In any lists, you can find the position of an element using the functions indexOf() and lastIndexOf(). They return the first and the last position of an element equal to the given argument in the list. If there are no such elements, both functions return -1.

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4, 2, 5)
```

```kotlin
    println(numbers.indexOf(2))
    println(numbers.lastIndexOf(2))
//sampleEnd
}
```

There is also a pair of functions that take a predicate and search for elements matching it:

- indexOfFirst() returns the index of the first element matching the predicate or -1 if there are no such elements.
- indexOfLast() returns the index of the last element matching the predicate or -1 if there are no such elements.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers.indexOfFirst { it > 2})
    println(numbers.indexOfLast { it % 2 == 1})
//sampleEnd
}
```

Binary search in sorted lists

There is one more way to search elements in lists – binary search. It works significantly faster than other built-in search functions but requires the list to be sorted in ascending order according to a certain ordering: natural or another one provided in the function parameter. Otherwise, the result is undefined.

To search an element in a sorted list, call the binarySearch() function passing the value as an argument. If such an element exists, the function returns its index; otherwise, it returns (-insertionPoint - 1) where insertionPoint is the index where this element should be inserted so that the list remains sorted. If there is more than one element with the given value, the search can return any of their indices.

You can also specify an index range to search in: in this case, the function searches only between two provided indices.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.sort()
    println(numbers)
    println(numbers.binarySearch("two"))  // 3
    println(numbers.binarySearch("z")) // -5
    println(numbers.binarySearch("two", 0, 2))  // -3
//sampleEnd
}
```

Comparator binary search

When list elements aren't Comparable, you should provide a Comparator to use in the binary search. The list must be sorted in ascending order according to this Comparator. Let's have a look at an example:

```kotlin
data class Product(val name: String, val price: Double)

fun main() {
//sampleStart
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch(Product("AppCode", 99.0), compareBy<Product> { it.price }.thenBy { it.name }))
//sampleEnd
}
```

Here's a list of Product instances that aren't Comparable and a Comparator that defines the order: product p1 precedes product p2 if p1's price is less than p2's price. So, having a list sorted ascending according to this order, we use binarySearch() to find the index of the specified Product.

Custom comparators are also handy when a list uses an order different from natural one, for example, a case-insensitive order for String elements.

```kotlin
fun main() {
```

```kotlin
//sampleStart
    val colors = listOf("Blue", "green", "ORANGE", "Red", "yellow")
    println(colors.binarySearch("RED", String.CASE_INSENSITIVE_ORDER)) // 3
//sampleEnd
}
```

Comparison binary search

Binary search with comparison function lets you find elements without providing explicit search values. Instead, it takes a comparison function mapping elements to Int values and searches for the element where the function returns zero. The list must be sorted in the ascending order according to the provided function; in other words, the return values of comparison must grow from one list element to the next one.

```kotlin
import kotlin.math.sign
//sampleStart
data class Product(val name: String, val price: Double)

fun priceComparison(product: Product, price: Double) = sign(product.price - price).toInt()

fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch { priceComparison(it, 99.0) })
}
//sampleEnd
```

Both comparator and comparison binary search can be performed for list ranges as well.

List write operations

In addition to the collection modification operations described in Collection write operations, mutable lists support specific write operations. Such operations use the index to access elements to broaden the list modification capabilities.

Add

To add elements to a specific position in a list, use add() and addAll() providing the position for element insertion as an additional argument. All elements that come after the position shift to the right.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "five", "six")
    numbers.add(1, "two")
    numbers.addAll(2, listOf("three", "four"))
    println(numbers)
//sampleEnd
}
```

Update

Lists also offer a function to replace an element at a given position - set() and its operator form []. set() doesn't change the indexes of other elements.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "five", "three")
    numbers[1] =  "two"
    println(numbers)
//sampleEnd
}
```

fill() simply replaces all the collection elements with the specified value.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.fill(3)
    println(numbers)
//sampleEnd
}
```

Remove

To remove an element at a specific position from a list, use the removeAt() function providing the position as an argument. All indices of elements that come after the element being removed will decrease by one.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.removeAt(1)
    println(numbers)
//sampleEnd
}
```

Sort

In Collection Ordering, we describe operations that retrieve collection elements in specific orders. For mutable lists, the standard library offers similar extension functions that perform the same ordering operations in place. When you apply such an operation to a list instance, it changes the order of elements in that exact instance.

The in-place sorting functions have similar names to the functions that apply to read-only lists, but without the ed/d suffix:

- sort* instead of sorted* in the names of all sorting functions: sort(), sortDescending(), sortBy(), and so on.
- shuffle() instead of shuffled().
- reverse() instead of reversed().

asReversed() called on a mutable list returns another mutable list which is a reversed view of the original list. Changes in that view are reflected in the original list. The following example shows sorting functions for mutable lists:

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")

    numbers.sort()
    println("Sort into ascending: $numbers")
    numbers.sortDescending()
    println("Sort into descending: $numbers")

    numbers.sortBy { it.length }
    println("Sort into ascending by length: $numbers")
    numbers.sortByDescending { it.last() }
    println("Sort into descending by the last letter: $numbers")

    numbers.sortWith(compareBy<String> { it.length }.thenBy { it })
    println("Sort by Comparator: $numbers")

    numbers.shuffle()
    println("Shuffle: $numbers")

    numbers.reverse()
    println("Reverse: $numbers")
//sampleEnd
}
```

Set-specific operations

The Kotlin collections package contains extension functions for popular operations on sets: finding intersections, merging, or subtracting collections from each other.

To merge two collections into one, use the union() function. It can be used in the infix form a union b. Note that for ordered collections the order of the operands is important: in the resulting collection, the elements of the first operand go before the elements of the second.

To find an intersection between two collections (elements present in both of them), use intersect(). To find collection elements not present in another collection, use subtract(). Both these functions can be called in the infix form as well, for example, a intersect b.

```
fun main() {
//sampleStart
    val numbers = setOf("one", "two", "three")

    println(numbers union setOf("four", "five"))
    println(setOf("four", "five") union numbers)

    println(numbers intersect setOf("two", "one"))
    println(numbers subtract setOf("three", "four"))
    println(numbers subtract setOf("four", "three")) // same output
//sampleEnd
}
```

You can also apply union, intersect, and subtract to List. However, their result is always a Set, even on lists. In this result, all the duplicate elements are merged into one and the index access is not available.

```
fun main() {
//sampleStart
    val list1 = listOf(1, 1, 2 ,3, 5, 8, -1)
    val list2 = listOf(1, 1, 2, 2 ,3, 5)
    println(list1 intersect list2) // result on two lists is a Set
    println(list1 union list2)     // equal elements are merged into one
//sampleEnd
}
```

Map-specific operations

In maps, types of both keys and values are user-defined. Key-based access to map entries enables various map-specific processing capabilities from getting a value by key to separate filtering of keys and values. On this page, we provide descriptions of the map processing functions from the standard library.

Retrieve keys and values

For retrieving a value from a map, you must provide its key as an argument of the get() function. The shorthand [key] syntax is also supported. If the given key is not found, it returns null. There is also the function getValue() which has slightly different behavior: it throws an exception if the key is not found in the map. Additionally, you have two more options to handle the key absence:

- getOrElse() works the same way as for lists: the values for non-existent keys are returned from the given lambda function.
- getOrDefault() returns the specified default value if the key is not found.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.get("one"))
    println(numbersMap["one"])
    println(numbersMap.getOrDefault("four", 10))
    println(numbersMap["five"])               // null
    //numbersMap.getValue("six")      // exception!
//sampleEnd
}
```

To perform operations on all keys or all values of a map, you can retrieve them from the properties keys and values accordingly. keys is a set of all map keys and values is a collection of all map values.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.keys)
    println(numbersMap.values)
//sampleEnd
```

```
}
```

Filter

You can filter maps with the filter() function as well as other collections. When calling filter() on a map, pass to it a predicate with a Pair as an argument. This enables you to use both the key and the value in the filtering predicate.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
    println(filteredMap)
//sampleEnd
}
```

There are also two specific ways for filtering maps: by keys and by values. For each way, there is a function: filterKeys() and filterValues(). Both return a new map of entries which match the given predicate. The predicate for filterKeys() checks only the element keys, the one for filterValues() checks only values.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
    val filteredValuesMap = numbersMap.filterValues { it < 10 }

    println(filteredKeysMap)
    println(filteredValuesMap)
//sampleEnd
}
```

Plus and minus operators

Due to the key access to elements, plus (+) and minus (-) operators work for maps differently than for other collections. plus returns a Map that contains elements of its both operands: a Map on the left and a Pair or another Map on the right. When the right-hand side operand contains entries with keys present in the left-hand side Map, the result map contains the entries from the right side.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap + Pair("four", 4))
    println(numbersMap + Pair("one", 10))
    println(numbersMap + mapOf("five" to 5, "one" to 11))
//sampleEnd
}
```

minus creates a Map from entries of a Map on the left except those with keys from the right-hand side operand. So, the right-hand side operand can be either a single key or a collection of keys: list, set, and so on.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap - "one")
    println(numbersMap - listOf("two", "four"))
//sampleEnd
}
```

For details on using plusAssign (+=) and minusAssign (-=) operators on mutable maps, see Map write operations below.

Map write operations

Mutable maps offer map-specific write operations. These operations let you change the map content using the key-based access to the values.

There are certain rules that define write operations on maps:

- Values can be updated. In turn, keys never change: once you add an entry, its key is constant.
- For each key, there is always a single value associated with it. You can add and remove whole entries.

Below are descriptions of the standard library functions for write operations available on mutable maps.

Add and update entries

To add a new key-value pair to a mutable map, use put(). When a new entry is put into a LinkedHashMap (the default map implementation), it is added so that it comes last when iterating the map. In sorted maps, the positions of new elements are defined by the order of their keys.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    println(numbersMap)
//sampleEnd
}
```

To add multiple entries at a time, use putAll(). Its argument can be a Map or a group of Pairs: Iterable, Sequence, or Array.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap.putAll(setOf("four" to 4, "five" to 5))
    println(numbersMap)
//sampleEnd
}
```

Both put() and putAll() overwrite the values if the given keys already exist in the map. Thus, you can use them to update values of map entries.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    val previousValue = numbersMap.put("one", 11)
    println("value associated with 'one', before: $previousValue, after: ${numbersMap["one"]}")
    println(numbersMap)
//sampleEnd
}
```

You can also add new entries to maps using the shorthand operator form. There are two ways:

- plusAssign (+=) operator.
- the [] operator alias for set().

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap["three"] = 3     // calls numbersMap.put("three", 3)
    numbersMap += mapOf("four" to 4, "five" to 5)
    println(numbersMap)
//sampleEnd
}
```

When called with the key present in the map, operators overwrite the values of the corresponding entries.

Remove entries

To remove an entry from a mutable map, use the remove() function. When calling remove(), you can pass either a key or a whole key-value-pair. If you specify both the key and value, the element with this key will be removed only if its value matches the second argument.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap.remove("one")
    println(numbersMap)
    numbersMap.remove("three", 4)            //doesn't remove anything
    println(numbersMap)
//sampleEnd
}
```

You can also remove entries from a mutable map by their keys or values. To do this, call remove() on the map's keys or values providing the key or the value of an entry. When called on values, remove() removes only the first entry with the given value.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain" to 3)
    numbersMap.keys.remove("one")
    println(numbersMap)
    numbersMap.values.remove(3)
    println(numbersMap)
//sampleEnd
}
```

The minusAssign (-=) operator is also available for mutable maps.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap -= "two"
    println(numbersMap)
    numbersMap -= "five"             //doesn't remove anything
    println(numbersMap)
//sampleEnd
}
```

Scope functions

The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object. When you call such a function on an object with a lambda expression provided, it forms a temporary scope. In this scope, you can access the object without its name. Such functions are called scope functions. There are five of them: let, run, with, apply, and also.

Basically, these functions all perform the same action: execute a block of code on an object. What's different is how this object becomes available inside the block and what is the result of the whole expression.

Here's a typical example of how to use a scope function:

```
data class Person(var name: String, var age: Int, var city: String) {
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
//sampleStart
    Person("Alice", 20, "Amsterdam").let {
        println(it)
        it.moveTo("London")
        it.incrementAge()
        println(it)
    }
//sampleEnd
}
```

If you write the same without let, you'll have to introduce a new variable and repeat its name whenever you use it.

```
data class Person(var name: String, var age: Int, var city: String) {
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}
```

```kotlin
fun main() {
//sampleStart
    val alice = Person("Alice", 20, "Amsterdam")
    println(alice)
    alice.moveTo("London")
    alice.incrementAge()
    println(alice)
//sampleEnd
}
```

Scope functions don't introduce any new technical capabilities, but they can make your code more concise and readable.

Due to the many similarities between scope functions, choosing the right one for your use case can be tricky. The choice mainly depends on your intent and the consistency of use in your project. Below, we provide detailed descriptions of the differences between scope functions and their conventions.

Function selection

To help you choose the right scope function for your purpose, we provide this table that summarizes the key differences between them.

Function	Object reference	Return value	Is extension function
let	it	Lambda result	Yes
run	this	Lambda result	Yes
run	-	Lambda result	No: called without the context object
with	this	Lambda result	No: takes the context object as an argument.
apply	this	Context object	Yes
also	it	Context object	Yes

Detailed information about these functions is provided in the dedicated sections below.

Here is a short guide for choosing scope functions depending on the intended purpose:

- Executing a lambda on non-null objects: let
- Introducing an expression as a variable in local scope: let
- Object configuration: apply
- Object configuration and computing the result: run
- Running statements where an expression is required: non-extension run
- Additional effects: also
- Grouping function calls on an object: with

The use cases of different scope functions overlap, so you can choose which functions to use based on the specific conventions used in your project or team.

Although scope functions can make your code more concise, avoid overusing them: it can make your code hard to read and lead to errors. We also recommend that you avoid nesting scope functions and be careful when chaining them because it's easy to get confused about the current context object and value of this or it.

Distinctions

Because scope functions are similar in nature, it's important to understand the differences between them. There are two main differences between each scope function:

- The way they refer to the context object.
- Their return value.

Context object: this or it

Inside the lambda passed to a scope function, the context object is available by a short reference instead of its actual name. Each scope function uses one of two ways to reference the context object: as a lambda receiver (this) or as a lambda argument (it). Both provide the same capabilities, so we describe the pros and cons of each for different use cases and provide recommendations for their use.

```
fun main() {
    val str = "Hello"
    // this
    str.run {
        println("The string's length: $length")
        //println("The string's length: ${this.length}") // does the same
    }

    // it
    str.let {
        println("The string's length is ${it.length}")
    }
}
```

this

run, with, and apply reference the context object as a lambda receiver - by keyword this. Hence, in their lambdas, the object is available as it would be in ordinary class functions.

In most cases, you can omit this when accessing the members of the receiver object, making the code shorter. On the other hand, if this is omitted, it can be hard to distinguish between the receiver members and external objects or functions. So having the context object as a receiver (this) is recommended for lambdas that mainly operate on the object's members by calling its functions or assigning values to properties.

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
//sampleStart
    val adam = Person("Adam").apply {
        age = 20                       // same as this.age = 20
        city = "London"
    }
    println(adam)
//sampleEnd
}
```

it

In turn, let and also reference the context object as a lambda argument. If the argument name is not specified, the object is accessed by the implicit default name it. it is shorter than this and expressions with it are usually easier to read.

However, when calling the object's functions or properties you don't have the object available implicitly like this. Hence, accessing the context object via it is better when the object is mostly used as an argument in function calls. it is also better if you use multiple variables in the code block.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
            writeToLog("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
```

```
    println(i)
//sampleEnd
}
```

The example below demonstrates referencing the context object as a lambda argument with argument name: value.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also { value ->
            writeToLog("getRandomInt() generated value $value")
        }
    }

    val i = getRandomInt()
    println(i)
//sampleEnd
}
```

Return value

Scope functions differ by the result they return:

- apply and also return the context object.
- let, run, and with return the lambda result.

You should consider carefully what return value you want based on what you want to do next in your code. This helps you to choose the best scope function to use.

Context object

The return value of apply and also is the context object itself. Hence, they can be included into call chains as side steps: you can continue chaining function calls on the same object, one after another.

```
fun main() {
//sampleStart
    val numberList = mutableListOf<Double>()
    numberList.also { println("Populating the list") }
        .apply {
            add(2.71)
            add(3.14)
            add(1.0)
        }
        .also { println("Sorting the list") }
        .sort()
//sampleEnd
    println(numberList)
}
```

They also can be used in return statements of functions returning the context object.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
            writeToLog("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
//sampleEnd
}
```

Lambda result

let, run, and with return the lambda result. So you can use them when assigning the result to a variable, chaining operations on the result, and so on.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    val countEndsWithE = numbers.run {
        add("four")
        add("five")
        count { it.endsWith("e") }
    }
    println("There are $countEndsWithE elements that end with e.")
//sampleEnd
}
```

Additionally, you can ignore the return value and use a scope function to create a temporary scope for local variables.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        val firstItem = first()
        val lastItem = last()
        println("First item: $firstItem, last item: $lastItem")
    }
//sampleEnd
}
```

Functions

To help you choose the right scope function for your use case, we describe them in detail and provide recommendations for use. Technically, scope functions are interchangeable in many cases, so the examples show conventions for using them.

let

- The context object is available as an argument (it).
- The return value is the lambda result.

let can be used to invoke one or more functions on results of call chains. For example, the following code prints the results of two operations on a collection:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    val resultList = numbers.map { it.length }.filter { it > 3 }
    println(resultList)
//sampleEnd
}
```

With let, you can rewrite the above example so that you're not assigning the result of the list operations to a variable:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.map { it.length }.filter { it > 3 }.let {
        println(it)
        // and more function calls if needed
    }
//sampleEnd
}
```

If the code block passed to let contains a single function with it as an argument, you can use the method reference (::) instead of the lambda argument:

```
fun main() {
//sampleStart
```

```kotlin
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.map { it.length }.filter { it > 3 }.let(::println)
//sampleEnd
}
```

let is often used to execute a code block containing non-null values. To perform actions on a non-null object, use the safe call operator ?. on it and call let with the actions in its lambda.

```kotlin
fun processNonNullString(str: String) {}

fun main() {
//sampleStart
    val str: String? = "Hello"
    //processNonNullString(str)       // compilation error: str can be null
    val length = str?.let {
        println("let() called on $it")
        processNonNullString(it)      // OK: 'it' is not null inside '?.let { }'
        it.length
    }
//sampleEnd
}
```

You can also use let to introduce local variables with a limited scope to make your code easier to read. To define a new variable for the context object, provide its name as the lambda argument so that it can be used instead of the default it.

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val modifiedFirstItem = numbers.first().let { firstItem ->
        println("The first item of the list is '$firstItem'")
        if (firstItem.length >= 5) firstItem else "!" + firstItem + "!"
    }.uppercase()
    println("First item after modifications: '$modifiedFirstItem'")
//sampleEnd
}
```

with

- The context object is available as a receiver (this).
- The return value is the lambda result.

As with is not an extension function: the context object is passed as an argument, but inside the lambda, it's available as a receiver (this).

We recommend using with for calling functions on the context object when you don't need to use the returned result. In code, with can be read as "with this object, do the following."

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        println("'with' is called with argument $this")
        println("It contains $size elements")
    }
//sampleEnd
}
```

You can also use with to introduce a helper object whose properties or functions are used for calculating a value.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    val firstAndLast = with(numbers) {
        "The first element is ${first()}," +
        " the last element is ${last()}"
    }
    println(firstAndLast)
//sampleEnd
}
```

run

- The context object is available as a receiver (this).
- The return value is the lambda result.

run does the same as with but it is implemented as an extension function. So like let, you can call it on the context object using dot notation.

run is useful when your lambda both initializes objects and computes the return value.

```
class MultiportService(var url: String, var port: Int) {
    fun prepareRequest(): String = "Default request"
    fun query(request: String): String = "Result for query '$request'"
}

fun main() {
//sampleStart
    val service = MultiportService("https://example.kotlinlang.org", 80)

    val result = service.run {
        port = 8080
        query(prepareRequest() + " to port $port")
    }

    // the same code written with let() function:
    val letResult = service.let {
        it.port = 8080
        it.query(it.prepareRequest() + " to port ${it.port}")
    }
//sampleEnd
    println(result)
    println(letResult)
}
```

You can also invoke run as a non-extension function. The non-extension variant of run has no context object, but it still returns the lambda result. Non-extension run lets you execute a block of several statements where an expression is required.

```
fun main() {
//sampleStart
    val hexNumberRegex = run {
        val digits = "0-9"
        val hexDigits = "A-Fa-f"
        val sign = "+-"

        Regex("[$sign]?[$digits$hexDigits]+")
    }

    for (match in hexNumberRegex.findAll("+123 -FFFF !%*& 88 XYZ")) {
        println(match.value)
    }
//sampleEnd
}
```

apply

- The context object is available as a receiver (this).
- The return value is the object itself.

As apply returns the context object itself, we recommend that you use it for code blocks that don't return a value and that mainly operate on the members of the receiver object. The most common use case for apply is for object configuration. Such calls can be read as "apply the following assignments to the object."

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
//sampleStart
    val adam = Person("Adam").apply {
        age = 32
        city = "London"
    }
    println(adam)
//sampleEnd
}
```

Another use case for apply is to include apply in multiple call chains for more complex processing.

also

- The context object is available as an argument (it).
- The return value is the object itself.

also is useful for performing some actions that take the context object as an argument. Use also for actions that need a reference to the object rather than its properties and functions, or when you don't want to shadow the this reference from an outer scope.

When you see also in code, you can read it as "and also do the following with the object."

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    numbers
        .also { println("The list elements before adding new one: $it") }
        .add("four")
//sampleEnd
}
```

takeIf and takeUnless

In addition to scope functions, the standard library contains the functions takeIf and takeUnless. These functions let you embed checks of an object's state in call chains.

When called on an object along with a predicate, takeIf returns this object if it satisfies the given predicate. Otherwise, it returns null. So, takeIf is a filtering function for a single object.

takeUnless has the opposite logic of takeIf. When called on an object along with a predicate, takeUnless returns null if it satisfies the given predicate. Otherwise, it returns the object.

When using takeIf or takeUnless, the object is available as a lambda argument (it).

```
import kotlin.random.*

fun main() {
//sampleStart
    val number = Random.nextInt(100)

    val evenOrNull = number.takeIf { it % 2 == 0 }
    val oddOrNull = number.takeUnless { it % 2 == 0 }
    println("even: $evenOrNull, odd: $oddOrNull")
//sampleEnd
}
```

> When chaining other functions after takeIf and takeUnless, don't forget to perform a null check or use a safe call (?.) because their return value is nullable.

```
fun main() {
//sampleStart
    val str = "Hello"
    val caps = str.takeIf { it.isNotEmpty() }?.uppercase()
   //val caps = str.takeIf { it.isNotEmpty() }.uppercase() //compilation error
    println(caps)
//sampleEnd
}
```

takeIf and takeUnless are especially useful in combination with scope functions. For example, you can chain takeIf and takeUnless with let to run a code block on objects that match the given predicate. To do this, call takeIf on the object and then call let with a safe call (?). For objects that don't match the predicate, takeIf returns null and let isn't invoked.

```
fun main() {
```

```kotlin
//sampleStart
    fun displaySubstringPosition(input: String, sub: String) {
        input.indexOf(sub).takeIf { it >= 0 }?.let {
            println("The substring $sub is found in $input.")
            println("Its start position is $it.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
//sampleEnd
}
```

For comparison, below is an example of how the same function can be written without using takeIf or scope functions:

```kotlin
fun main() {
//sampleStart
    fun displaySubstringPosition(input: String, sub: String) {
        val index = input.indexOf(sub)
        if (index >= 0) {
            println("The substring $sub is found in $input.")
            println("Its start position is $index.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
//sampleEnd
}
```

Opt-in requirements

The Kotlin standard library provides a mechanism for requiring and giving explicit consent for using certain elements of APIs. This mechanism lets library developers inform users of their APIs about specific conditions that require opt-in, for example, if an API is in the experimental state and is likely to change in the future.

To prevent potential issues, the compiler warns users of such APIs about these conditions and requires them to opt in before using the API.

Opt in to using API

If a library author marks a declaration from a library's API as requiring opt-in, you should give an explicit consent for using it in your code. There are several ways to opt in to such APIs, all applicable without technical limitations. You are free to choose the way that you find best for your situation.

Propagating opt-in

When you use an API in the code intended for third-party use (a library), you can propagate its opt-in requirement to your API as well. To do this, annotate your declaration with the opt-in requirement annotation of the API used in its body. This enables you to use API elements that require opt-in.

```kotlin
// Library code
@RequiresOptIn(message = "This API is experimental. It may be changed in the future without notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // Opt-in requirement annotation

@MyDateTime
class DateProvider // A class requiring opt-in
```

```kotlin
// Client code
fun getYear(): Int {
    val dateProvider: DateProvider // Error: DateProvider requires opt-in
    // ...
}

@MyDateTime
fun getDate(): Date {
    val dateProvider: DateProvider // OK: the function requires opt-in as well
    // ...
}
```

```
fun displayDate() {
    println(getDate()) // Error: getDate() requires opt-in
}
```

As you can see in this example, the annotated function appears to be a part of the @MyDateTime API. So, such an opt-in propagates the opt-in requirement to the client code; its clients will see the same warning message and be required to consent as well.

Implicit usages of APIs that require opt-in also require opt-in. If an API element doesn't have an opt-in requirement annotation but its signature includes a type declared as requiring opt-in, its usage will still raise a warning. See the example below.

```
// Client code
fun getDate(dateProvider: DateProvider): Date { // Error: DateProvider requires opt-in
    // ...
}

fun displayDate() {
    println(getDate()) // Warning: the signature of getDate() contains DateProvider, which requires opt-in
}
```

To use multiple APIs that require opt-in, mark the declaration with all their opt-in requirement annotations.

Non-propagating opt-in

In modules that don't expose their own API, such as applications, you can opt in to using APIs without propagating the opt-in requirement to your code. In this case, mark your declaration with @OptIn passing the opt-in requirement annotation as its argument:

```
// Library code
@RequiresOptIn(message = "This API is experimental. It may be changed in the future without notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // Opt-in requirement annotation

@MyDateTime
class DateProvider // A class requiring opt-in
```

```
// Client code
@OptIn(MyDateTime::class)
fun getDate(): Date { // Uses DateProvider; doesn't propagate the opt-in requirement
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    println(getDate()) // OK: opt-in is not required
}
```

When somebody calls the function getDate(), they won't be informed about the opt-in requirements for APIs used in its body.

Note that if @OptIn applies to the declaration whose signature contains a type declared as requiring opt-in, the opt-in will still propagate:

```
// Client code
@OptIn(MyDateTime::class)
fun getDate(dateProvider: DateProvider): Date { // Has DateProvider as a part of a signature; propagates the opt-in requirement
    // ...
}

fun displayDate() {
    println(getDate()) // Warning: getDate() requires opt-in
}
```

To use an API that requires opt-in in all functions and classes in a file, add the file-level annotation @file:OptIn to the top of the file before the package specification and imports.

```
// Client code
@file:OptIn(MyDateTime::class)
```

Module-wide opt-in

> The -opt-in compiler option is available since Kotlin 1.6.0. For earlier Kotlin versions, use -Xopt-in.

If you don't want to annotate every usage of APIs that require opt-in, you can opt in to them for your whole module. To opt in to using an API in a module, compile it with the argument -opt-in, specifying the fully qualified name of the opt-in requirement annotation of the API you use: -opt-in=org.mylibrary.OptInAnnotation. Compiling with this argument has the same effect as if every declaration in the module had the annotation@OptIn(OptInAnnotation::class).

If you build your module with Gradle, you can add arguments like this:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
    compilerOptions.freeCompilerArgs.add("-opt-in=org.mylibrary.OptInAnnotation")
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        freeCompilerArgs.add("-opt-in=org.mylibrary.OptInAnnotation")
    }
}
```

If your Gradle module is a multiplatform module, use the optIn method:

Kotlin

```kotlin
sourceSets {
    all {
        languageSettings.optIn("org.mylibrary.OptInAnnotation")
    }
}
```

Groovy

```groovy
sourceSets {
    all {
        languageSettings {
            optIn('org.mylibrary.OptInAnnotation')
        }
    }
}
```

For Maven, it would be:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>...</executions>
            <configuration>
                <args>
                    <arg>-opt-in=org.mylibrary.OptInAnnotation</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

To opt in to multiple APIs on the module level, add one of the described arguments for each opt-in requirement marker used in your module.

Require opt-in for API

Create opt-in requirement annotations

If you want to require explicit consent to using your module's API, create an annotation class to use as an opt-in requirement annotation. This class must be annotated with @RequiresOptIn:

```
@RequiresOptIn
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime
```

Opt-in requirement annotations must meet several requirements:

- BINARY or RUNTIME retention
- No EXPRESSION, FILE, TYPE, or TYPE_PARAMETER among targets
- No parameters.

An opt-in requirement can have one of two severity levels:

- RequiresOptIn.Level.ERROR. Opt-in is mandatory. Otherwise, the code that uses marked API won't compile. Default level.
- RequiresOptIn.Level.WARNING. Opt-in is not mandatory, but advisable. Without it, the compiler raises a warning.

To set the desired level, specify the level parameter of the @RequiresOptIn annotation.

Additionally, you can provide a message to inform API users about special condition of using the API. The compiler will show it to users that use the API without opt-in.

```
@RequiresOptIn(level = RequiresOptIn.Level.WARNING, message = "This API is experimental. It can be incompatibly changed in the
future.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime
```

If you publish multiple independent features that require opt-in, declare an annotation for each. This makes the use of API safer for your clients: they can use only the features that they explicitly accept. This also lets you remove the opt-in requirements from the features independently.

Mark API elements

To require an opt-in to using an API element, annotate its declaration with an opt-in requirement annotation:

```
@MyDateTime
class DateProvider

@MyDateTime
fun getTime(): Time {}
```

Note that for some language elements, an opt-in requirement annotation is not applicable:

- You cannot annotate a backing field or a getter of a property, just the property itself.
- You cannot annotate a local variable or a value parameter.

Opt-in requirements for pre-stable APIs

If you use opt-in requirements for features that are not stable yet, carefully handle the API graduation to avoid breaking the client code.

Once your pre-stable API graduates and is released in a stable state, remove its opt-in requirement annotations from declarations. The clients will be able to use

them without restriction. However, you should leave the annotation classes in modules so that the existing client code remains compatible.

To let the API users update their modules accordingly (remove the annotations from their code and recompile), mark the annotations as @Deprecated and provide the explanation in the deprecation message.

```
@Deprecated("This opt-in requirement is not used anymore. Remove its usages from your code.")
@RequiresOptIn
annotation class ExperimentalDateTime
```

Coroutines guide

Kotlin provides only minimal low-level APIs in its standard library to enable other libraries to utilize coroutines. Unlike many other languages with similar capabilities, async and await are not keywords in Kotlin and are not even part of its standard library. Moreover, Kotlin's concept of suspending function provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.

kotlinx.coroutines is a rich library for coroutines developed by JetBrains. It contains a number of high-level coroutine-enabled primitives that this guide covers, including launch, async, and others.

This is a guide about the core features of kotlinx.coroutines with a series of examples, divided up into different topics.

In order to use coroutines as well as follow the examples in this guide, you need to add a dependency on the kotlinx-coroutines-core module as explained in the project README.

Table of contents

- Coroutines basics
- Hands-on: Intro to coroutines and channels
- Cancellation and timeouts
- Composing suspending functions
- Coroutine context and dispatchers
- Asynchronous Flow
- Channels
- Coroutine exceptions handling
- Shared mutable state and concurrency
- Select expression (experimental)
- Tutorial: Debug coroutines using IntelliJ IDEA
- Tutorial: Debug Kotlin Flow using IntelliJ IDEA

Additional references

- Guide to UI programming with coroutines
- Coroutines design document (KEEP)
- Full kotlinx.coroutines API reference
- Best practices for coroutines in Android
- Additional Android resources for Kotlin coroutines and flow

Coroutines basics

This section covers basic coroutine concepts.

Your first coroutine

A coroutine is an instance of suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.

Coroutines can be thought of as light-weight threads, but there is a number of important differences that make their real-life usage very different from threads.

Run the following code to get to your first working coroutine:

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is delayed
}
//sampleEnd
```

> You can get the full code here.

You will see the following result:

```
Hello
World!
```

Let's dissect what this code does.

launch is a coroutine builder. It launches a new coroutine concurrently with the rest of the code, which continues to work independently. That's why Hello has been printed first.

delay is a special suspending function. It suspends the coroutine for a specific time. Suspending a coroutine does not block the underlying thread, but allows other coroutines to run and use the underlying thread for their code.

runBlocking is also a coroutine builder that bridges the non-coroutine world of a regular fun main() and the code with coroutines inside of runBlocking { ... } curly braces. This is highlighted in an IDE by this: CoroutineScope hint right after the runBlocking opening curly brace.

If you remove or forget runBlocking in this code, you'll get an error on the launch call, since launch is declared only on the CoroutineScope:

```
Unresolved reference: launch
```

The name of runBlocking means that the thread that runs it (in this case — the main thread) gets blocked for the duration of the call, until all the coroutines inside runBlocking { ... } complete their execution. You will often see runBlocking used like that at the very top-level of the application and quite rarely inside the real code, as threads are expensive resources and blocking them is inefficient and is often not desired.

Structured concurrency
Coroutines follow a principle of structured concurrency which means that new coroutines can only be launched in a specific CoroutineScope which delimits the lifetime of the coroutine. The above example shows that runBlocking establishes the corresponding scope and that is why the previous example waits until World! is printed after a second's delay and only then exits.

In a real application, you will be launching a lot of coroutines. Structured concurrency ensures that they are not lost and do not leak. An outer scope cannot complete until all its children coroutines complete. Structured concurrency also ensures that any errors in the code are properly reported and are never lost.

Extract function refactoring

Let's extract the block of code inside launch { ... } into a separate function. When you perform "Extract function" refactoring on this code, you get a new function with the suspend modifier. This is your first suspending function. Suspending functions can be used inside coroutines just like regular functions, but their additional feature is that they can, in turn, use other suspending functions (like delay in this example) to suspend execution of a coroutine.

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking { // this: CoroutineScope
    launch { doWorld() }
    println("Hello")
}

// this is your first suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
//sampleEnd
```

You can get the full code here.

Scope builder

In addition to the coroutine scope provided by different builders, it is possible to declare your own scope using the coroutineScope builder. It creates a coroutine scope and does not complete until all launched children complete.

runBlocking and coroutineScope builders may look similar because they both wait for their body and all its children to complete. The main difference is that the runBlocking method blocks the current thread for waiting, while coroutineScope just suspends, releasing the underlying thread for other usages. Because of that difference, runBlocking is a regular function and coroutineScope is a suspending function.

You can use coroutineScope from any suspending function. For example, you can move the concurrent printing of Hello and World into a suspend fun doWorld() function:

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking {
    doWorld()
}

suspend fun doWorld() = coroutineScope {  // this: CoroutineScope
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
//sampleEnd
```

You can get the full code here.

This code also prints:

```
Hello
World!
```

Scope builder and concurrency

A coroutineScope builder can be used inside any suspending function to perform multiple concurrent operations. Let's launch two concurrent coroutines inside a doWorld suspending function:

```
import kotlinx.coroutines.*
```

```
//sampleStart
// Sequentially executes doWorld followed by "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}

// Concurrently executes both sections
suspend fun doWorld() = coroutineScope { // this: CoroutineScope
    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
//sampleEnd
```

You can get the full code here.

Both pieces of code inside launch { ... } blocks execute concurrently, with World 1 printed first, after a second from start, and World 2 printed next, after two seconds from start. A coroutineScope in doWorld completes only after both are complete, so doWorld returns and allows Done string to be printed only after that:

```
Hello
World 1
World 2
Done
```

An explicit job

A launch coroutine builder returns a Job object that is a handle to the launched coroutine and can be used to explicitly wait for its completion. For example, you can wait for completion of the child coroutine and then print "Done" string:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch { // launch a new coroutine and keep a reference to its Job
        delay(1000L)
        println("World!")
    }
    println("Hello")
    job.join() // wait until child coroutine completes
    println("Done")
//sampleEnd
}
```

You can get the full code here.

This code produces:

```
Hello
World!
Done
```

Coroutines are light-weight

Coroutines are less resource-intensive than JVM threads. Code that exhausts the JVM's available memory when using threads can be expressed using coroutines without hitting resource limits. For example, the following code launches 50,000 distinct coroutines that each waits 5 seconds and then prints a period ('.') while

consuming very little memory:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(50_000) { // launch a lot of coroutines
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

> You can get the full code here.

If you write the same program using threads (remove runBlocking, replace launch with thread, and replace delay with Thread.sleep), it will consume a lot of memory. Depending on your operating system, JDK version, and its settings, it will either throw an out-of-memory error or start threads slowly so that there are never too many concurrently running threads.

Coroutines and channels – tutorial

In this tutorial, you'll learn how to use coroutines in IntelliJ IDEA to perform network requests without blocking the underlying thread or callbacks.

> No prior knowledge of coroutines is required, but you're expected to be familiar with basic Kotlin syntax.

You'll learn:

- Why and how to use suspending functions to perform network requests.
- How to send requests concurrently using coroutines.
- How to share information between different coroutines using channels.

For network requests, you'll need the Retrofit library, but the approach shown in this tutorial works similarly for any other libraries that support coroutines.

> You can find solutions for all of the tasks on the solutions branch of the project's repository.

Before you start

1. Download and install the latest version of IntelliJ IDEA.
2. Clone the project template by choosing Get from VCS on the Welcome screen or selecting File | New | Project from Version Control.

   You can also clone it from the command line:

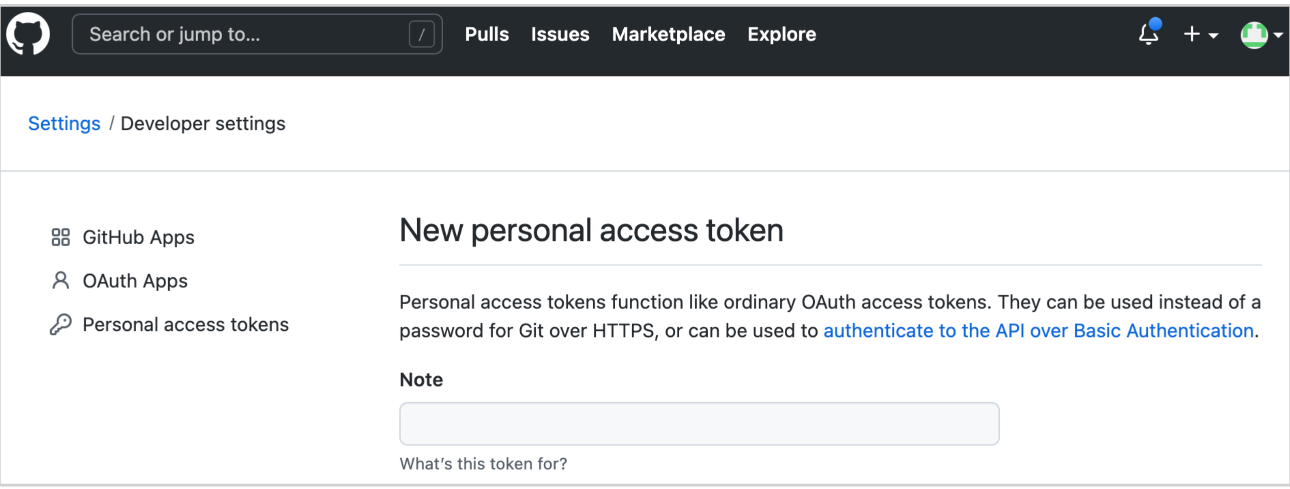
   ```
   git clone https://github.com/kotlin-hands-on/intro-coroutines
   ```

Generate a GitHub developer token

You'll be using the GitHub API in your project. To get access, provide your GitHub account name and either a password or a token. If you have two-factor authentication enabled, a token will be enough.

Generate a new GitHub token to use the GitHub API with your account:

1. Specify the name of your token, for example, coroutines-tutorial:



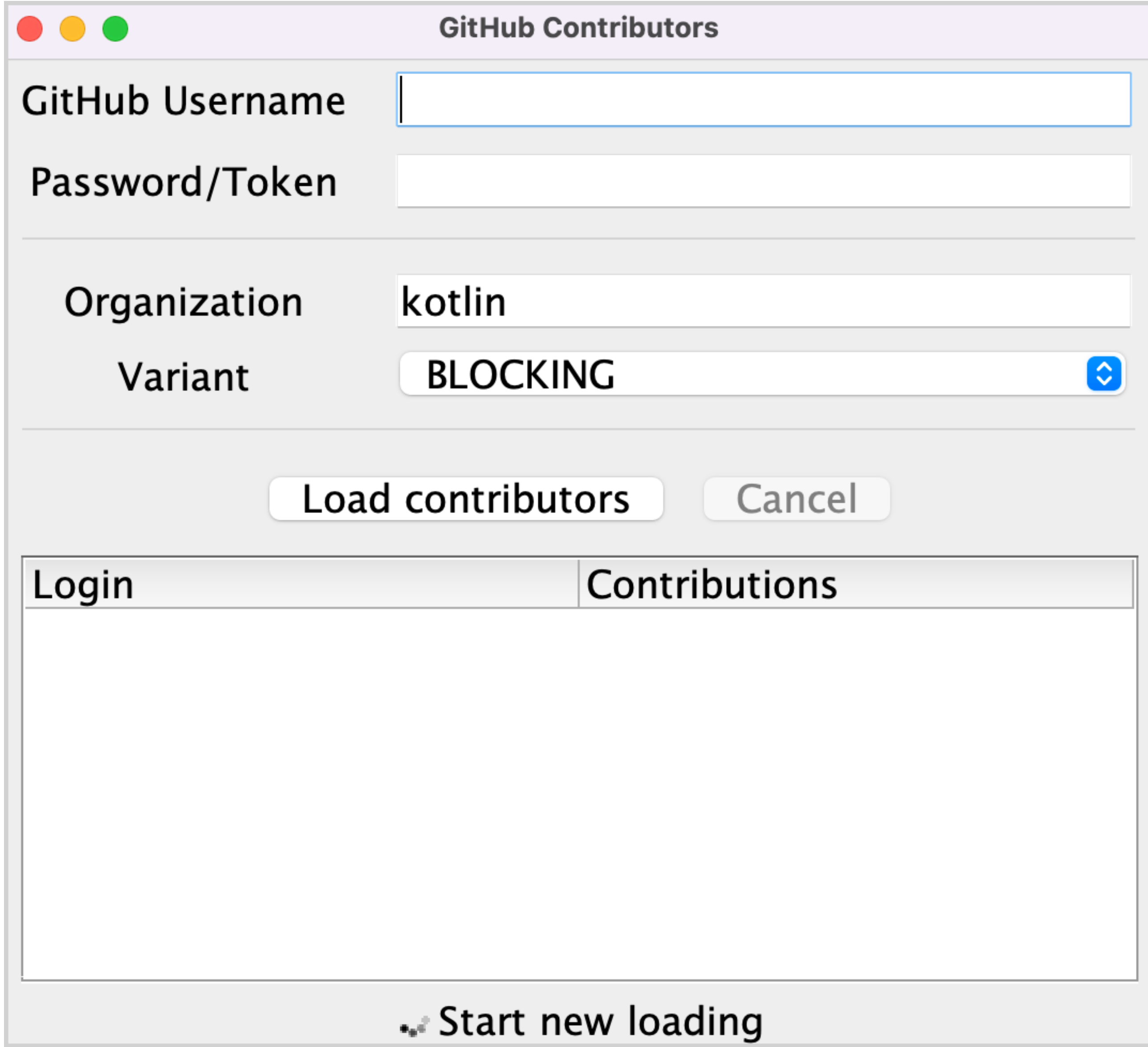
Generate a new GitHub token

2. Do not select any scopes. Click Generate token at the bottom of the page.
3. Copy the generated token.

Run the code

The program loads the contributors for all of the repositories under the given organization (named “kotlin” by default). Later you'll add logic to sort the users by the number of their contributions.

1. Open the src/contributors/main.kt file and run the main() function. You'll see the following window:



First window

If the font is too small, adjust it by changing the value of setDefaultFontSize(18f) in the main() function.

2. Provide your GitHub username and token (or password) in the corresponding fields.
3. Make sure that the BLOCKING option is selected in the Variant dropdown menu.
4. Click Load contributors. The UI should freeze for some time and then show the list of contributors.
5. Open the program output to ensure the data has been loaded. The list of contributors is logged after each successful request.

There are different ways of implementing this logic: by using blocking requests or callbacks. You'll compare these solutions with one that uses coroutines and see how channels can be used to share information between different coroutines.

Blocking requests

You will use the Retrofit library to perform HTTP requests to GitHub. It allows requesting the list of repositories under the given organization and the list of contributors for each repository:

```
interface GitHubService {
    @GET("orgs/{org}/repos?per_page=100")
```

```kotlin
    fun getOrgReposCall(
        @Path("org") org: String
    ): Call<List<Repo>>

    @GET("repos/{owner}/{repo}/contributors?per_page=100")
    fun getRepoContributorsCall(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Call<List<User>>
}
```

This API is used by the loadContributorsBlocking() function to fetch the list of contributors for the given organization.

1. Open src/tasks/Request1Blocking.kt to see its implementation:

```kotlin
fun loadContributorsBlocking(service: GitHubService, req: RequestData): List<User> {
    val repos = service
        .getOrgReposCall(req.org)   // #1
        .execute()                  // #2
        .also { logRepos(req, it) } // #3
        .body() ?: emptyList()      // #4

    return repos.flatMap { repo ->
        service
            .getRepoContributorsCall(req.org, repo.name) // #1
            .execute()                                   // #2
            .also { logUsers(repo, it) }                 // #3
            .bodyList()                                  // #4
    }.aggregate()
}
```

- At first, you get a list of the repositories under the given organization and store it in the repos list. Then for each repository, the list of contributors is requested, and all of the lists are merged into one final list of contributors.
- getOrgReposCall() and getRepoContributorsCall() both return an instance of the *Call class (#1). At this point, no request is sent.
- *Call.execute() is then invoked to perform the request (#2). execute() is a synchronous call that blocks the underlying thread.
- When you get the response, the result is logged by calling the specific logRepos() and logUsers() functions (#3). If the HTTP response contains an error, this error will be logged here.
- Finally, get the response's body, which contains the data you need. For this tutorial, you'll use an empty list as a result in case there is an error, and you'll log the corresponding error (#4).

2. To avoid repeating .body() ?: emptyList(), an extension function bodyList() is declared:

```kotlin
fun <T> Response<List<T>>.bodyList(): List<T> {
    return body() ?: emptyList()
}
```

3. Run the program again and take a look at the system output in IntelliJ IDEA. It should have something like this:

```
1770 [AWT-EventQueue-0] INFO  Contributors - kotlin: loaded 40 repos
2025 [AWT-EventQueue-0] INFO  Contributors - kotlin-examples: loaded 23 contributors
2229 [AWT-EventQueue-0] INFO  Contributors - kotlin-koans: loaded 45 contributors
...
```

- The first item on each line is the number of milliseconds that have passed since the program started, then the thread name in square brackets. You can see from which thread the loading request is called.
- The final item on each line is the actual message: how many repositories or contributors were loaded.

This log output demonstrates that all of the results were logged from the main thread. When you run the code with a BLOCKING option, the window freezes and doesn't react to input until the loading is finished. All of the requests are executed from the same thread as the one called loadContributorsBlocking() is from, which is the main UI thread (in Swing, it's an AWT event dispatching thread). This main thread becomes blocked, and that's why the UI is frozen:

The blocked main thread

After the list of contributors has loaded, the result is updated.

4. In src/contributors/Contributors.kt, find the loadContributors() function responsible for choosing how the contributors are loaded and look at how loadContributorsBlocking() is called:

```
when (getSelectedVariant()) {
    BLOCKING -> { // Blocking UI thread
        val users = loadContributorsBlocking(service, req)
        updateResults(users, startTime)
    }
}
```

- The updateResults() call goes right after the loadContributorsBlocking() call.
- updateResults() updates the UI, so it must always be called from the UI thread.
- Since loadContributorsBlocking() is also called from the UI thread, the UI thread becomes blocked and the UI is frozen.

Task 1

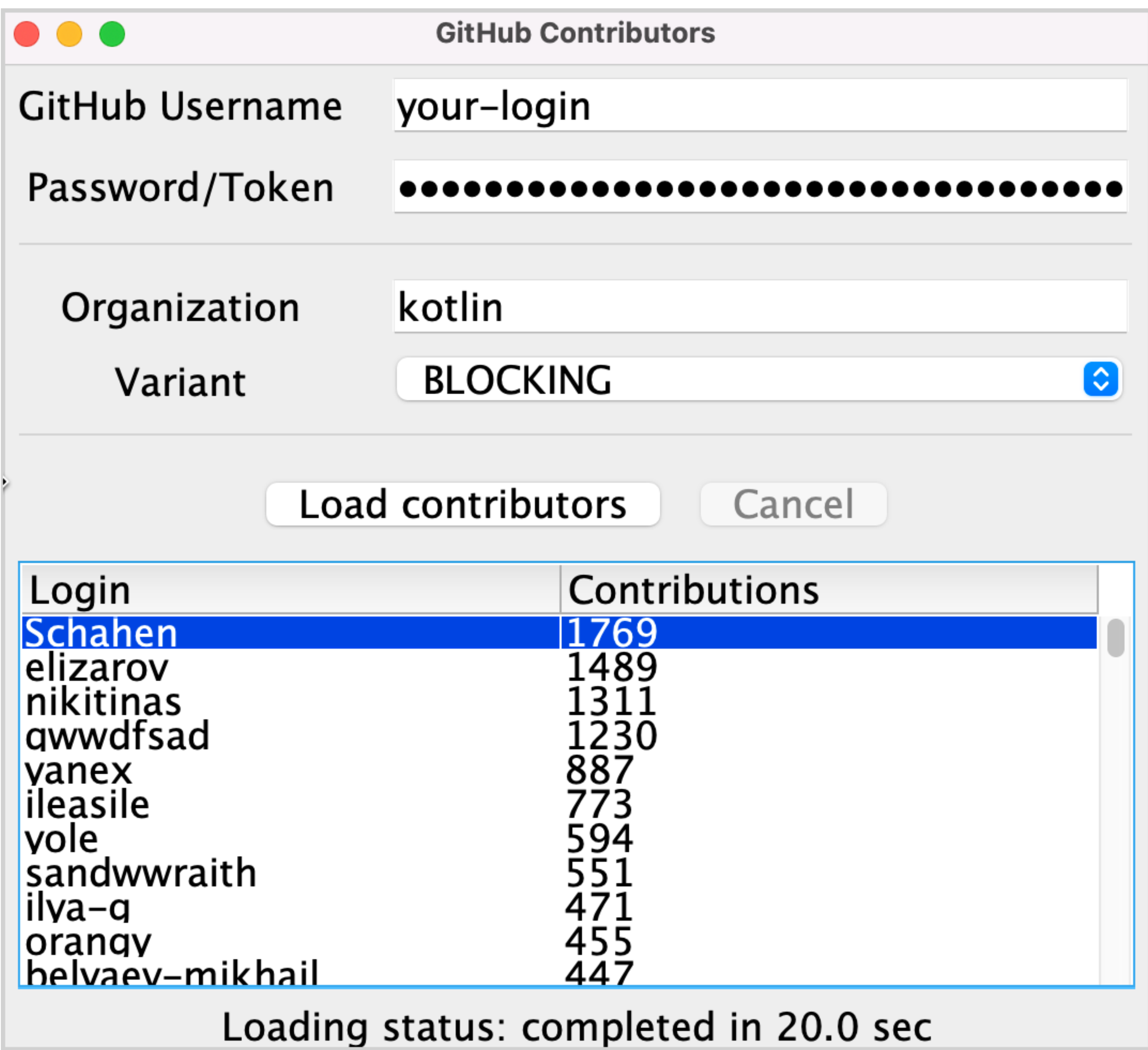
The first task helps you familiarize yourself with the task domain. Currently, each contributor's name is repeated several times, once for every project they have taken part in. Implement the aggregate() function combining the users so that each contributor is added only once. The User.contributions property should contain the total number of contributions of the given user to all the projects. The resulting list should be sorted in descending order according to the number of contributions.

Open src/tasks/Aggregation.kt and implement the List<User>.aggregate() function. Users should be sorted by the total number of their contributions.

The corresponding test file test/tasks/AggregationKtTest.kt shows an example of the expected result.

> You can jump between the source code and the test class automatically by using the IntelliJ IDEA shortcut Ctrl+Shift+T / ⇧ ⌘ T.

After implementing this task, the resulting list for the "kotlin" organization should be similar to the following:



The list for the "kotlin" organization

Solution for task 1

1. To group users by login, use groupBy(), which returns a map from a login to all occurrences of the user with this login in different repositories.
2. For each map entry, count the total number of contributions for each user and create a new instance of the User class by the given name and total of contributions.
3. Sort the resulting list in descending order:

```
fun List<User>.aggregate(): List<User> =
    groupBy { it.login }
        .map { (login, group) -> User(login, group.sumOf { it.contributions }) }
        .sortedByDescending { it.contributions }
```

An alternative solution is to use the groupingBy() function instead of groupBy().

Callbacks

The previous solution works, but it blocks the thread and therefore freezes the UI. A traditional approach that avoids this is to use callbacks.

Instead of calling the code that should be invoked right after the operation is completed, you can extract it into a separate callback, often a lambda, and pass that lambda to the caller in order for it to be called later.

To make the UI responsive, you can either move the whole computation to a separate thread or switch to the Retrofit API which uses callbacks instead of blocking calls.

Use a background thread

1. Open src/tasks/Request2Background.kt and see its implementation. First, the whole computation is moved to a different thread. The thread() function starts a new thread:

```
thread {
    loadContributorsBlocking(service, req)
}
```

   Now that all of the loading has been moved to a separate thread, the main thread is free and can be occupied by other tasks:

The freed main thread

2. The signature of the loadContributorsBackground() function changes. It takes an updateResults() callback as the last argument to call it after all the loading completes:

```
fun loadContributorsBackground(
    service: GitHubService, req: RequestData,
    updateResults: (List<User>) -> Unit
)
```

3. Now when the loadContributorsBackground() is called, the updateResults() call goes in the callback, not immediately afterward as it did before:

```
loadContributorsBackground(service, req) { users ->
    SwingUtilities.invokeLater {
        updateResults(users, startTime)
    }
}
```

   By calling SwingUtilities.invokeLater, you ensure that the updateResults() call, which updates the results, happens on the main UI thread (AWT event dispatching thread).

However, if you try to load the contributors via the BACKGROUND option, you can see that the list is updated but nothing changes.

Task 2

Fix the loadContributorsBackground() function in src/tasks/Request2Background.kt so that the resulting list is shown in the UI.

Solution for task 2

If you try to load the contributors, you can see in the log that the contributors are loaded but the result isn't displayed. To fix this, call updateResults() on the resulting list of users:

```
thread {
    updateResults(loadContributorsBlocking(service, req))
}
```

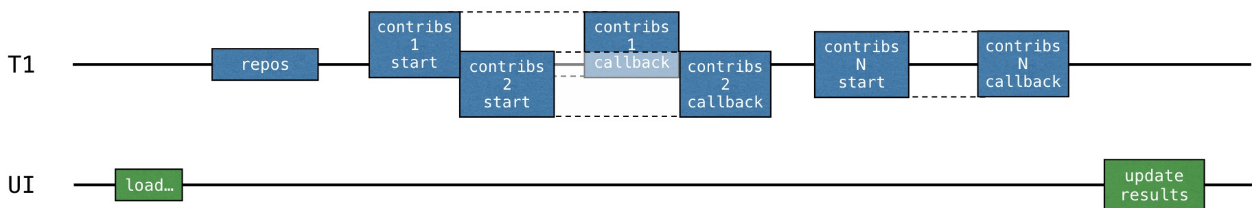
Make sure to call the logic passed in the callback explicitly. Otherwise, nothing will happen.

Use the Retrofit callback API

In the previous solution, the whole loading logic is moved to the background thread, but that still isn't the best use of resources. All of the loading requests go sequentially and the thread is blocked while waiting for the loading result, while it could have been occupied by other tasks. Specifically, the thread could start loading another request to receive the entire result earlier.

Handling the data for each repository should then be divided into two parts: loading and processing the resulting response. The second processing part should be extracted into a callback.

The loading for each repository can then be started before the result for the previous repository is received (and the corresponding callback is called):



Using callback API

The Retrofit callback API can help achieve this. The Call.enqueue() function starts an HTTP request and takes a callback as an argument. In this callback, you need to specify what needs to be done after each request.

Open src/tasks/Request3Callbacks.kt and see the implementation of loadContributorsCallbacks() that uses this API:

```
fun loadContributorsCallbacks(
    service: GitHubService, req: RequestData,
    updateResults: (List<User>) -> Unit
) {
    service.getOrgReposCall(req.org).onResponse { responseRepos ->  // #1
        logRepos(req, responseRepos)
        val repos = responseRepos.bodyList()

        val allUsers = mutableListOf<User>()
        for (repo in repos) {
            service.getRepoContributorsCall(req.org, repo.name)
                .onResponse { responseUsers ->  // #2
                    logUsers(repo, responseUsers)
                    val users = responseUsers.bodyList()
                    allUsers += users
                }
            }
        }
        // TODO: Why doesn't this code work? How to fix that?
        updateResults(allUsers.aggregate())
    }
```

- For convenience, this code fragment uses the onResponse() extension function declared in the same file. It takes a lambda as an argument rather than an object expression.
- The logic for handling the responses is extracted into callbacks: the corresponding lambdas start at lines #1 and #2.

However, the provided solution doesn't work. If you run the program and load contributors by choosing the CALLBACKS option, you'll see that nothing is shown. However, the tests that immediately return the result pass.

Think about why the given code doesn't work as expected and try to fix it, or see the solutions below.

Task 3 (optional)

Rewrite the code in the src/tasks/Request3Callbacks.kt file so that the loaded list of contributors is shown.

The first attempted solution for task 3

In the current solution, many requests are started concurrently, which decreases the total loading time. However, the result isn't loaded. This is because the updateResults() callback is called right after all of the loading requests are started, before the allUsers list has been filled with the data.

You could try to fix this with a change like the following:

```
val allUsers = mutableListOf<User>()
for ((index, repo) in repos.withIndex()) {   // #1
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            logUsers(repo, responseUsers)
            val users = responseUsers.bodyList()
            allUsers += users
            if (index == repos.lastIndex) {    // #2
                updateResults(allUsers.aggregate())
            }
        }
}
```

- First, you iterate over the list of repos with an index (#1).
- Then, from each callback, you check whether it's the last iteration (#2).
- And if that's the case, the result is updated.

However, this code also fails to achieve our objective. Try to find the answer yourself, or see the solution below.

The second attempted solution for task 3

Since the loading requests are started concurrently, there's no guarantee that the result for the last one comes last. The results can come in any order.

Thus, if you compare the current index with the lastIndex as a condition for completion, you risk losing the results for some repos.

If the request that processes the last repo returns faster than some prior requests (which is likely to happen), all of the results for requests that take more time will be lost.

One way to fix this is to introduce an index and check whether all of the repositories have already been processed:

```
val allUsers = Collections.synchronizedList(mutableListOf<User>())
val numberOfProcessed = AtomicInteger()
for (repo in repos) {
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            logUsers(repo, responseUsers)
            val users = responseUsers.bodyList()
            allUsers += users
            if (numberOfProcessed.incrementAndGet() == repos.size) {
                updateResults(allUsers.aggregate())
            }
        }
}
```

This code uses a synchronized version of the list and AtomicInteger() because, in general, there's no guarantee that different callbacks that process getRepoContributors() requests will always be called from the same thread.

The third attempted solution for task 3

An even better solution is to use the CountDownLatch class. It stores a counter initialized with the number of repositories. This counter is decremented after processing each repository. It then waits until the latch is counted down to zero before updating the results:

```
val countDownLatch = CountDownLatch(repos.size)
for (repo in repos) {
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            // processing repository
            countDownLatch.countDown()
        }
}
countDownLatch.await()
updateResults(allUsers.aggregate())
```

The result is then updated from the main thread. This is more direct than delegating the logic to the child threads.

After reviewing these three attempts at a solution, you can see that writing correct code with callbacks is non-trivial and error-prone, especially when several underlying threads and synchronization occur.

As an additional exercise, you can implement the same logic using a reactive approach with the RxJava library. All of the necessary dependencies and solutions for using RxJava can be found in a separate rx branch. It is also possible to complete this tutorial and implement or check the proposed Rx versions for a proper comparison.

Suspending functions

You can implement the same logic using suspending functions. Instead of returning Call<List<Repo>>, define the API call as a suspending function as follows:

```
interface GitHubService {
    @GET("orgs/{org}/repos?per_page=100")
    suspend fun getOrgRepos(
        @Path("org") org: String
    ): List<Repo>
}
```

- getOrgRepos() is defined as a suspend function. When you use a suspending function to perform a request, the underlying thread isn't blocked. More details about how this works will come in later sections.
- getOrgRepos() returns the result directly instead of returning a Call. If the result is unsuccessful, an exception is thrown.

Alternatively, Retrofit allows returning the result wrapped in Response. In this case, the result body is provided, and it is possible to check for errors manually. This tutorial uses the versions that return Response.

In src/contributors/GitHubService.kt, add the following declarations to the GitHubService interface:

```
interface GitHubService {
    // getOrgReposCall & getRepoContributorsCall declarations

    @GET("orgs/{org}/repos?per_page=100")
    suspend fun getOrgRepos(
        @Path("org") org: String
    ): Response<List<Repo>>

    @GET("repos/{owner}/{repo}/contributors?per_page=100")
    suspend fun getRepoContributors(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Response<List<User>>
}
```

Task 4

Your task is to change the code of the function that loads contributors to make use of two new suspending functions, getOrgRepos() and getRepoContributors(). The new loadContributorsSuspend() function is marked as suspend to use the new API.

Suspending functions can't be called everywhere. Calling a suspending function from loadContributorsBlocking() will result in an error with the message "Suspend function 'getOrgRepos' should be called only from a coroutine or another suspend function".

1. Copy the implementation of loadContributorsBlocking() that is defined in src/tasks/Request1Blocking.kt into the loadContributorsSuspend() that is defined in src/tasks/Request4Suspend.kt.
2. Modify the code so that the new suspending functions are used instead of the ones that return Calls.
3. Run the program by choosing the SUSPEND option and ensure that the UI is still responsive while the GitHub requests are performed.

Solution for task 4

Replace .getOrgReposCall(req.org).execute() with .getOrgRepos(req.org) and repeat the same replacement for the second "contributors" request:

```
suspend fun loadContributorsSuspend(service: GitHubService, req: RequestData): List<User> {
    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()
```

```
    return repos.flatMap { repo ->
        service.getRepoContributors(req.org, repo.name)
            .also { logUsers(repo, it) }
            .bodyList()
    }.aggregate()
}
```

- loadContributorsSuspend() should be defined as a suspend function.
- You no longer need to call execute, which returned the Response before, because now the API functions return the Response directly. Note that this detail is specific to the Retrofit library. With other libraries, the API will be different, but the concept is the same.

Coroutines

The code with suspending functions looks similar to the "blocking" version. The major difference from the blocking version is that instead of blocking the thread, the coroutine is suspended:

```
block -> suspend
thread -> coroutine
```

> Coroutines are often called lightweight threads because you can run code on coroutines, similar to how you run code on threads. The operations that were blocking before (and had to be avoided) can now suspend the coroutine instead.

Starting a new coroutine

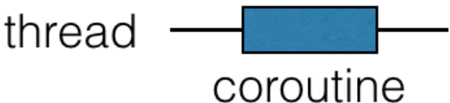
If you look at how loadContributorsSuspend() is used in src/contributors/Contributors.kt, you can see that it's called inside launch. launch is a library function that takes a lambda as an argument:

```
launch {
    val users = loadContributorsSuspend(req)
    updateResults(users, startTime)
}
```

Here launch starts a new computation that is responsible for loading the data and showing the results. The computation is suspendable – when performing network requests, it is suspended and releases the underlying thread. When the network request returns the result, the computation is resumed.

Such a suspendable computation is called a coroutine. So, in this case, launch starts a new coroutine responsible for loading data and showing the results.

Coroutines run on top of threads and can be suspended. When a coroutine is suspended, the corresponding computation is paused, removed from the thread, and stored in memory. Meanwhile, the thread is free to be occupied by other tasks:

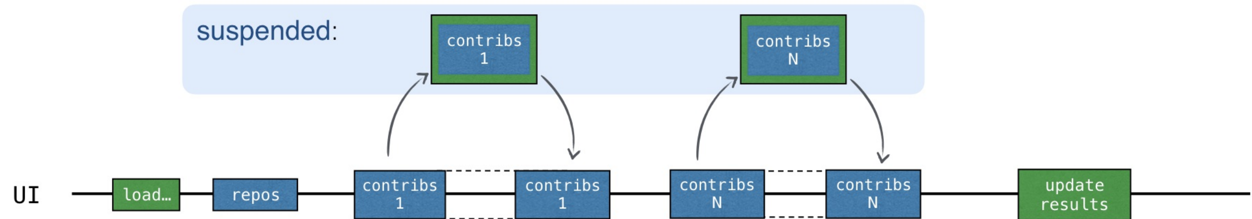


Suspending coroutines

When the computation is ready to be continued, it is returned to a thread (not necessarily the same one).

In the loadContributorsSuspend() example, each "contributors" request now waits for the result using the suspension mechanism. First, the new request is sent. Then, while waiting for the response, the whole "load contributors" coroutine that was started by the launch function is suspended.

The coroutine resumes only after the corresponding response is received:



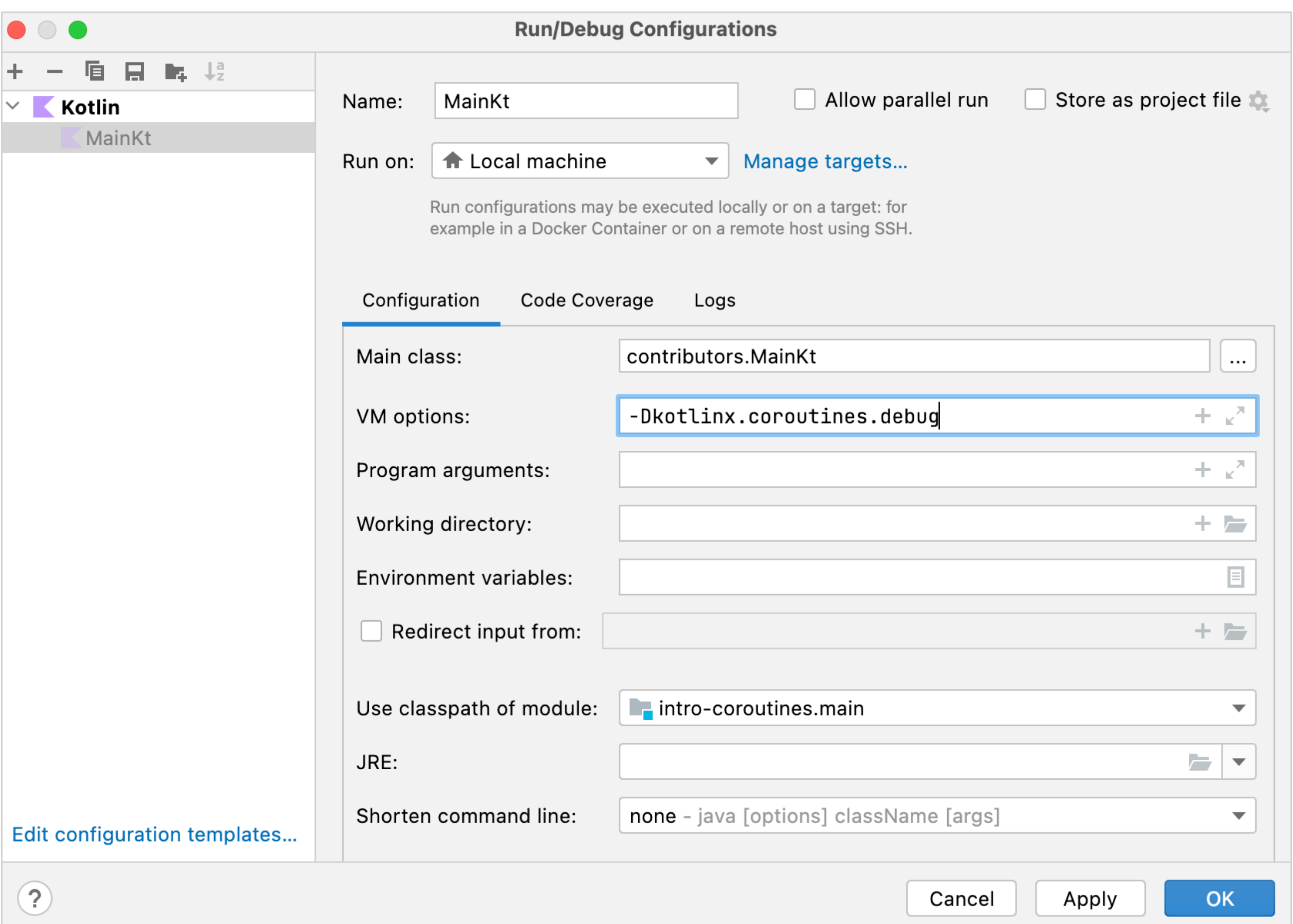
Suspending request

While the response is waiting to be received, the thread is free to be occupied by other tasks. The UI stays responsive, despite all the requests taking place on the main UI thread:

1. Run the program using the SUSPEND option. The log confirms that all of the requests are sent to the main UI thread:

```
2538 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin: loaded 30 repos
2729 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - ts2kt: loaded 11 contributors
3029 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin-koans: loaded 45 contributors
...
11252 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin-coroutines-workshop: loaded 1 contributors
```

2. The log can show you which coroutine the corresponding code is running on. To enable it, open Run | Edit configurations and add the -Dkotlinx.coroutines.debug VM option:



Edit run configuration

The coroutine name will be attached to the thread name while main() is run with this option. You can also modify the template for running all of the Kotlin files and enable this option by default.

Now all of the code runs on one coroutine, the "load contributors" coroutine mentioned above, denoted as @coroutine#1. While waiting for the result, you shouldn't reuse the thread for sending other requests because the code is written sequentially. The new request is sent only when the previous result is received.

Suspending functions treat the thread fairly and don't block it for "waiting". However, this doesn't yet bring any concurrency into the picture.

Concurrency

Kotlin coroutines are much less resource-intensive than threads. Each time you want to start a new computation asynchronously, you can create a new coroutine instead.

To start a new coroutine, use one of the main coroutine builders: launch, async, or runBlocking. Different libraries can define additional coroutine builders.

async starts a new coroutine and returns a Deferred object. Deferred represents a concept known by other names such as Future or Promise. It stores a computation, but it defers the moment you get the final result; it promises the result sometime in the future.

The main difference between async and launch is that launch is used to start a computation that isn't expected to return a specific result. launch returns a Job that represents the coroutine. It is possible to wait until it completes by calling Job.join().

Deferred is a generic type that extends Job. An async call can return a Deferred<Int> or a Deferred<CustomType>, depending on what the lambda returns (the last expression inside the lambda is the result).

To get the result of a coroutine, you can call await() on the Deferred instance. While waiting for the result, the coroutine that this await() is called from is suspended:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred: Deferred<Int> = async {
        loadData()
    }
    println("waiting...")
    println(deferred.await())
}

suspend fun loadData(): Int {
    println("loading...")
    delay(1000L)
    println("loaded!")
    return 42
}
```

runBlocking is used as a bridge between regular and suspending functions, or between the blocking and non-blocking worlds. It works as an adaptor for starting the top-level main coroutine. It is intended primarily to be used in main() functions and tests.

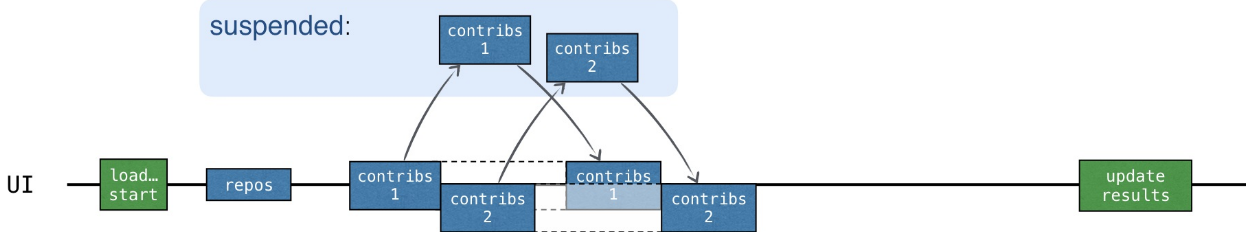
> Watch this video for a better understanding of coroutines.

If there is a list of deferred objects, you can call awaitAll() to await the results of all of them:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferreds: List<Deferred<Int>> = (1..3).map {
        async {
            delay(1000L * it)
            println("Loading $it")
            it
        }
    }
    val sum = deferreds.awaitAll().sum()
    println("$sum")
}
```

When each "contributors" request is started in a new coroutine, all of the requests are started asynchronously. A new request can be sent before the result for the

previous one is received:



Concurrent coroutines

The total loading time is approximately the same as in the CALLBACKS version, but it doesn't need any callbacks. What's more, async explicitly emphasizes which parts run concurrently in the code.

Task 5

In the Request5Concurrent.kt file, implement a loadContributorsConcurrent() function by using the previous loadContributorsSuspend() function.

Tip for task 5

You can only start a new coroutine inside a coroutine scope. Copy the content from loadContributorsSuspend() to the coroutineScope call so that you can call async functions there:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService,
    req: RequestData
): List<User> = coroutineScope {
    // ...
}
```

Base your solution on the following scheme:

```
val deferreds: List<Deferred<List<User>>> = repos.map { repo ->
    async {
        // load contributors for each repo
    }
}
deferreds.awaitAll() // List<List<User>>
```

Solution for task 5

Wrap each "contributors" request with async to create as many coroutines as there are repositories. async returns Deferred<List<User>>. This is not an issue because creating new coroutines is not very resource-intensive, so you can create as many as you need.

1. You can no longer use flatMap because the map result is now a list of Deferred objects, not a list of lists. awaitAll() returns List<List<User>>, so call flatten().aggregate() to get the result:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService,
    req: RequestData
): List<User> = coroutineScope {
    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    val deferreds: List<Deferred<List<User>>> = repos.map { repo ->
        async {
            service.getRepoContributors(req.org, repo.name)
                .also { logUsers(repo, it) }
                .bodyList()
        }
    }
    deferreds.awaitAll().flatten().aggregate()
```

```
}
```

2. Run the code and check the log. All of the coroutines still run on the main UI thread because multithreading hasn't been employed yet, but you can already see the benefits of running coroutines concurrently.

3. To change this code to run "contributors" coroutines on different threads from the common thread pool, specify Dispatchers.Default as the context argument for the async function:

```
async(Dispatchers.Default) { }
```

- CoroutineDispatcher determines what thread or threads the corresponding coroutine should be run on. If you don't specify one as an argument, async will use the dispatcher from the outer scope.

- Dispatchers.Default represents a shared pool of threads on the JVM. This pool provides a means for parallel execution. It consists of as many threads as there are CPU cores available, but it will still have two threads if there's only one core.

4. Modify the code in the loadContributorsConcurrent() function to start new coroutines on different threads from the common thread pool. Also, add additional logging before sending the request:

```
async(Dispatchers.Default) {
    log("starting loading for ${repo.name}")
    service.getRepoContributors(req.org, repo.name)
        .also { logUsers(repo, it) }
        .bodyList()
}
```

5. Run the program once again. In the log, you can see that each coroutine can be started on one thread from the thread pool and resumed on another:

```
1946 [DefaultDispatcher-worker-2 @coroutine#4] INFO  Contributors - starting loading for kotlin-koans
1946 [DefaultDispatcher-worker-3 @coroutine#5] INFO  Contributors - starting loading for dokka
1946 [DefaultDispatcher-worker-1 @coroutine#3] INFO  Contributors - starting loading for ts2kt
...
2178 [DefaultDispatcher-worker-1 @coroutine#4] INFO  Contributors - kotlin-koans: loaded 45 contributors
2569 [DefaultDispatcher-worker-1 @coroutine#5] INFO  Contributors - dokka: loaded 36 contributors
2821 [DefaultDispatcher-worker-2 @coroutine#3] INFO  Contributors - ts2kt: loaded 11 contributors
```

For instance, in this log excerpt, coroutine#4 is started on the worker-2 thread and continued on the worker-1 thread.

In src/contributors/Contributors.kt, check the implementation of the CONCURRENT option:

1. To run the coroutine only on the main UI thread, specify Dispatchers.Main as an argument:

```
launch(Dispatchers.Main) {
    updateResults()
}
```

- If the main thread is busy when you start a new coroutine on it, the coroutine becomes suspended and scheduled for execution on this thread. The coroutine will only resume when the thread becomes free.

- It's considered good practice to use the dispatcher from the outer scope rather than explicitly specifying it on each end-point. If you define loadContributorsConcurrent() without passing Dispatchers.Default as an argument, you can call this function in any context: with a Default dispatcher, with the main UI thread, or with a custom dispatcher.

- As you'll see later, when calling loadContributorsConcurrent() from tests, you can call it in the context with TestDispatcher, which simplifies testing. That makes this solution much more flexible.

2. To specify the dispatcher on the caller side, apply the following change to the project while letting loadContributorsConcurrent start coroutines in the inherited context:

```
launch(Dispatchers.Default) {
    val users = loadContributorsConcurrent(service, req)
    withContext(Dispatchers.Main) {
        updateResults(users, startTime)
    }
}
```

- updateResults() should be called on the main UI thread, so you call it with the context of Dispatchers.Main.

- withContext() calls the given code with the specified coroutine context, is suspended until it completes, and returns the result. An alternative but more verbose way to express this would be to start a new coroutine and explicitly wait (by suspending) until it completes: launch(context) { ... }.join().

3. Run the code and ensure that the coroutines are executed on the threads from the thread pool.

Structured concurrency

- The coroutine scope is responsible for the structure and parent-child relationships between different coroutines. New coroutines usually need to be started inside a scope.
- The coroutine context stores additional technical information used to run a given coroutine, like the coroutine custom name, or the dispatcher specifying the threads the coroutine should be scheduled on.

When launch, async, or runBlocking are used to start a new coroutine, they automatically create the corresponding scope. All of these functions take a lambda with a receiver as an argument, and CoroutineScope is the implicit receiver type:

```
launch { /* this: CoroutineScope */ }
```

- New coroutines can only be started inside a scope.
- launch and async are declared as extensions to CoroutineScope, so an implicit or explicit receiver must always be passed when you call them.
- The coroutine started by runBlocking is the only exception because runBlocking is defined as a top-level function. But because it blocks the current thread, it's intended primarily to be used in main() functions and tests as a bridge function.

A new coroutine inside runBlocking, launch, or async is started automatically inside the scope:

```
import kotlinx.coroutines.*

fun main() = runBlocking { /* this: CoroutineScope */
    launch { /* ... */ }
    // the same as:
    this.launch { /* ... */ }
}
```

When you call launch inside runBlocking, it's called as an extension to the implicit receiver of the CoroutineScope type. Alternatively, you could explicitly write this.launch.

The nested coroutine (started by launch in this example) can be considered as a child of the outer coroutine (started by runBlocking). This "parent-child" relationship works through scopes; the child coroutine is started from the scope corresponding to the parent coroutine.

It's possible to create a new scope without starting a new coroutine, by using the coroutineScope function. To start new coroutines in a structured way inside a suspend function without access to the outer scope, you can create a new coroutine scope that automatically becomes a child of the outer scope that this suspend function is called from. loadContributorsConcurrent()is a good example.

You can also start a new coroutine from the global scope using GlobalScope.async or GlobalScope.launch. This will create a top-level "independent" coroutine.

The mechanism behind the structure of the coroutines is called structured concurrency. It provides the following benefits over global scopes:

- The scope is generally responsible for child coroutines, whose lifetime is attached to the lifetime of the scope.
- The scope can automatically cancel child coroutines if something goes wrong or a user changes their mind and decides to revoke the operation.
- The scope automatically waits for the completion of all child coroutines. Therefore, if the scope corresponds to a coroutine, the parent coroutine does not complete until all the coroutines launched in its scope have completed.

When using GlobalScope.async, there is no structure that binds several coroutines to a smaller scope. Coroutines started from the global scope are all independent – their lifetime is limited only by the lifetime of the whole application. It's possible to store a reference to the coroutine started from the global scope and wait for its completion or cancel it explicitly, but that won't happen automatically as it would with structured concurrency.

Canceling the loading of contributors

Create two versions of the function that loads the list of contributors. Compare how both versions behave when you try to cancel the parent coroutine. The first version will use coroutineScope to start all of the child coroutines, whereas the second will use GlobalScope.

1. In Request5Concurrent.kt, add a 3-second delay to the loadContributorsConcurrent() function:

```kotlin
suspend fun loadContributorsConcurrent(
    service: GitHubService,
    req: RequestData
): List<User> = coroutineScope {
    // ...
    async {
        log("starting loading for ${repo.name}")
        delay(3000)
        // load repo contributors
    }
    // ...
}
```

The delay affects all of the coroutines that send requests, so that there's enough time to cancel the loading after the coroutines are started but before the requests are sent.

2. Create the second version of the loading function: copy the implementation of loadContributorsConcurrent() to loadContributorsNotCancellable() in Request5NotCancellable.kt and then remove the creation of a new coroutineScope.

3. The async calls now fail to resolve, so start them by using GlobalScope.async:

```kotlin
suspend fun loadContributorsNotCancellable(
    service: GitHubService,
    req: RequestData
): List<User> {   // #1
    // ...
    GlobalScope.async {   // #2
        log("starting loading for ${repo.name}")
        // load repo contributors
    }
    // ...
    return deferreds.awaitAll().flatten().aggregate()  // #3
}
```

- The function now returns the result directly, not as the last expression inside the lambda (lines #1 and #3).
- All of the "contributors" coroutines are started inside the GlobalScope, not as children of the coroutine scope (line #2).

4. Run the program and choose the CONCURRENT option to load the contributors.

5. Wait until all of the "contributors" coroutines are started, and then click Cancel. The log shows no new results, which means that all of the requests were indeed canceled:

```
2896 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin: loaded 40 repos
2901 [DefaultDispatcher-worker-2 @coroutine#4] INFO  Contributors - starting loading for kotlin-koans
...
2909 [DefaultDispatcher-worker-5 @coroutine#36] INFO  Contributors - starting loading for mpp-example
/* click on 'cancel' */
/* no requests are sent */
```

6. Repeat step 5, but this time choose the NOT_CANCELLABLE option:

```
2570 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin: loaded 30 repos
2579 [DefaultDispatcher-worker-1 @coroutine#4] INFO  Contributors - starting loading for kotlin-koans
...
2586 [DefaultDispatcher-worker-6 @coroutine#36] INFO  Contributors - starting loading for mpp-example
/* click on 'cancel' */
/* but all the requests are still sent: */
6402 [DefaultDispatcher-worker-5 @coroutine#4] INFO  Contributors - kotlin-koans: loaded 45 contributors
...
9555 [DefaultDispatcher-worker-8 @coroutine#36] INFO  Contributors - mpp-example: loaded 8 contributors
```

In this case, no coroutines are canceled, and all the requests are still sent.

7. Check how the cancellation is triggered in the "contributors" program. When the Cancel button is clicked, the main "loading" coroutine is explicitly canceled and the child coroutines are canceled automatically:

```kotlin
interface Contributors {

    fun loadContributors() {
        // ...
```

```kotlin
        when (getSelectedVariant()) {
            CONCURRENT -> {
                launch {
                    val users = loadContributorsConcurrent(service, req)
                    updateResults(users, startTime)
                }.setUpCancellation()      // #1
            }
        }
    }

    private fun Job.setUpCancellation() {
        val loadingJob = this              // #2

        // cancel the loading job if the 'cancel' button was clicked:
        val listener = ActionListener {
            loadingJob.cancel()            // #3
            updateLoadingStatus(CANCELED)
        }
        // add a listener to the 'cancel' button:
        addCancelListener(listener)

        // update the status and remove the listener
        // after the loading job is completed
    }
}
```

The launch function returns an instance of Job. Job stores a reference to the "loading coroutine", which loads all of the data and updates the results. You can call the setUpCancellation() extension function on it (line #1), passing an instance of Job as a receiver.

Another way you could express this would be to explicitly write:

```kotlin
val job = launch { }
job.setUpCancellation()
```

- For readability, you could refer to the setUpCancellation() function receiver inside the function with the new loadingJob variable (line #2).
- Then you could add a listener to the Cancel button so that when it's clicked, the loadingJob is canceled (line #3).

With structured concurrency, you only need to cancel the parent coroutine and this automatically propagates cancellation to all of the child coroutines.

Using the outer scope's context

When you start new coroutines inside the given scope, it's much easier to ensure that all of them run with the same context. It is also much easier to replace the context if needed.

Now it's time to learn how using the dispatcher from the outer scope works. The new scope created by the coroutineScope or by the coroutine builders always inherits the context from the outer scope. In this case, the outer scope is the scope the suspend loadContributorsConcurrent() function was called from:

```kotlin
launch(Dispatchers.Default) {  // outer scope
    val users = loadContributorsConcurrent(service, req)
    // ...
}
```

All of the nested coroutines are automatically started with the inherited context. The dispatcher is a part of this context. That's why all of the coroutines started by async are started with the context of the default dispatcher:

```kotlin
suspend fun loadContributorsConcurrent(
    service: GitHubService, req: RequestData
): List<User> = coroutineScope {
    // this scope inherits the context from the outer scope
    // ...
    async {   // nested coroutine started with the inherited context
        // ...
    }
    // ...
}
```

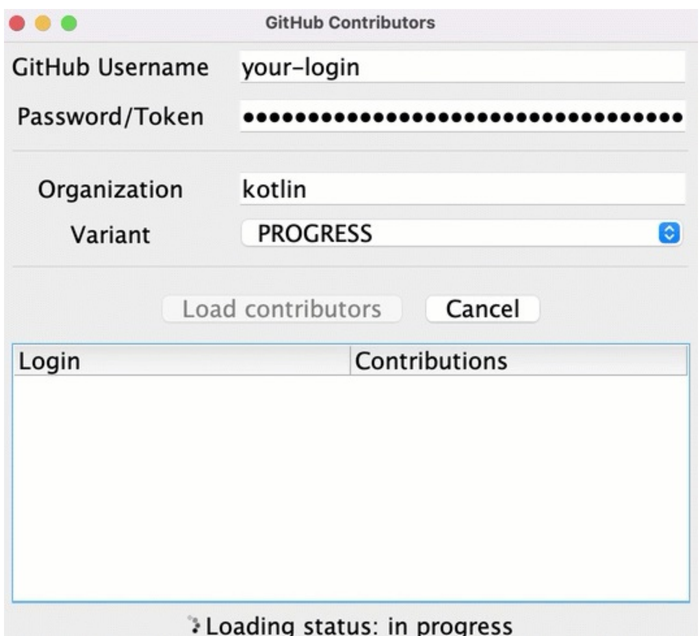
With structured concurrency, you can specify the major context elements (like dispatcher) once, when creating the top-level coroutine. All the nested coroutines then inherit the context and modify it only if needed.

When you write code with coroutines for UI applications, for example Android ones, it's a common practice to use CoroutineDispatchers.Main by default for the top coroutine and then to explicitly put a different dispatcher when you need to run the code on a different thread.

Showing progress

Despite the information for some repositories being loaded rather quickly, the user only sees the resulting list after all of the data has been loaded. Until then, the loader icon runs showing the progress, but there's no information about the current state or what contributors are already loaded.

You can show the intermediate results earlier and display all of the contributors after loading the data for each of the repositories:



Loading data

To implement this functionality, in the src/tasks/Request6Progress.kt, you'll need to pass the logic updating the UI as a callback, so that it's called on each intermediate state:

```
suspend fun loadContributorsProgress(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) {
    // loading the data
    // calling `updateResults()` on intermediate states
}
```

On the call site in Contributors.kt, the callback is passed to update the results from the Main thread for the PROGRESS option:

```
launch(Dispatchers.Default) {
    loadContributorsProgress(service, req) { users, completed ->
        withContext(Dispatchers.Main) {
            updateResults(users, startTime, completed)
        }
    }
}
```

- The updateResults() parameter is declared as suspend in loadContributorsProgress(). It's necessary to call withContext, which is a suspend function inside the corresponding lambda argument.
- updateResults() callback takes an additional Boolean parameter as an argument specifying whether the loading has completed and the results are final.

Task 6

In the Request6Progress.kt file, implement the loadContributorsProgress() function that shows the intermediate progress. Base it on the loadContributorsSuspend() function from Request4Suspend.kt.

- Use a simple version without concurrency; you'll add it later in the next section.
- The intermediate list of contributors should be shown in an "aggregated" state, not just the list of users loaded for each repository.
- The total number of contributions for each user should be increased when the data for each new repository is loaded.

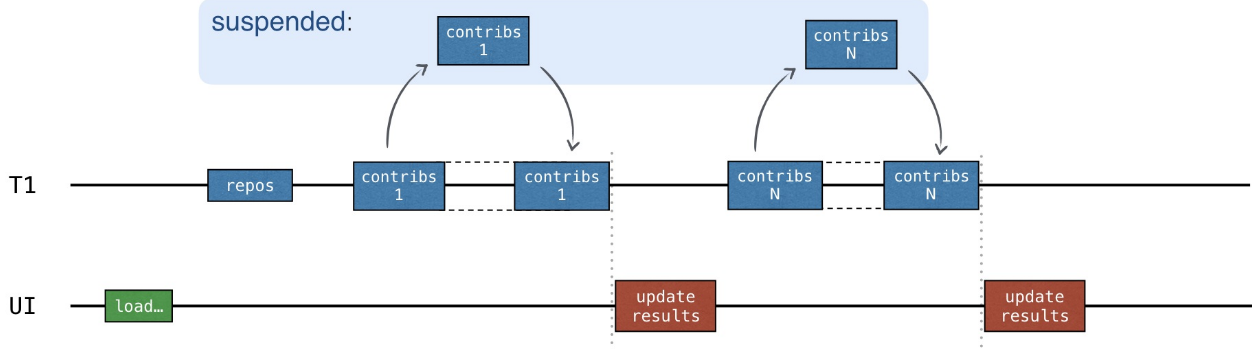
Solution for task 6

To store the intermediate list of loaded contributors in the "aggregated" state, define an allUsers variable which stores the list of users, and then update it after contributors for each new repository are loaded:

```
suspend fun loadContributorsProgress(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) {
    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    var allUsers = emptyList<User>()
    for ((index, repo) in repos.withIndex()) {
        val users = service.getRepoContributors(req.org, repo.name)
            .also { logUsers(repo, it) }
            .bodyList()

        allUsers = (allUsers + users).aggregate()
        updateResults(allUsers, index == repos.lastIndex)
    }
}
```

Consecutive vs concurrent

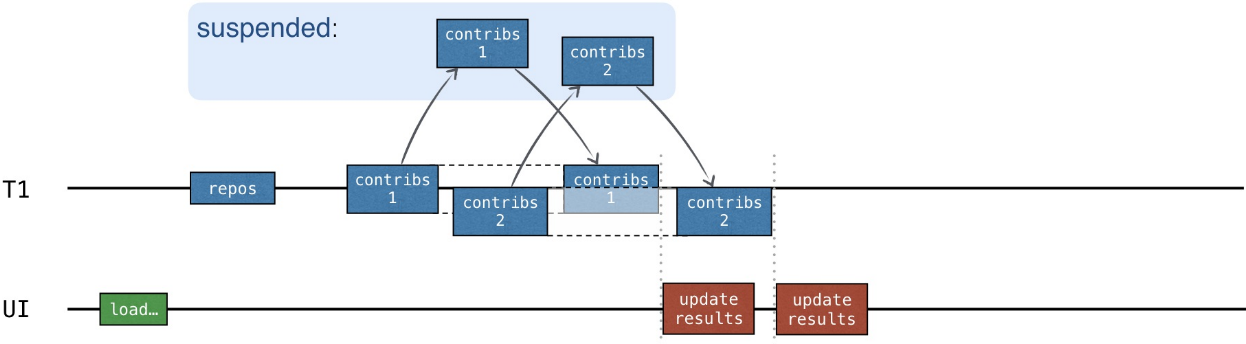
An updateResults() callback is called after each request is completed:



Progress on requests

This code doesn't include concurrency. It's sequential, so you don't need synchronization.

The best option would be to send requests concurrently and update the intermediate results after getting the response for each repository:



Concurrent requests

To add concurrency, use channels.

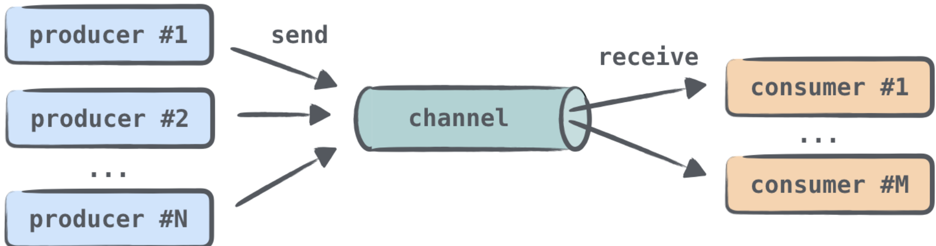
Channels

Writing code with a shared mutable state is quite difficult and error-prone (like in the solution using callbacks). A simpler way is to share information by communication rather than by using a common mutable state. Coroutines can communicate with each other through channels.

Channels are communication primitives that allow data to be passed between coroutines. One coroutine can send some information to a channel, while another can receive that information from it:

Using channels

A coroutine that sends (produces) information is often called a producer, and a coroutine that receives (consumes) information is called a consumer. One or multiple coroutines can send information to the same channel, and one or multiple coroutines can receive data from it:



Using channels with many coroutines

When many coroutines receive information from the same channel, each element is handled only once by one of the consumers. Once an element is handled, it is immediately removed from the channel.

You can think of a channel as similar to a collection of elements, or more precisely, a queue, in which elements are added to one end and received from the other. However, there's an important difference: unlike collections, even in their synchronized versions, a channel can suspend send()and receive() operations. This happens when the channel is empty or full. The channel can be full if the channel size has an upper bound.

Channel is represented by three different interfaces: SendChannel, ReceiveChannel, and Channel, with the latter extending the first two. You usually create a channel and give it to producers as a SendChannel instance so that only they can send information to the channel. You give a channel to consumers as a

ReceiveChannel instance so that only they can receive from it. Both send and receive methods are declared as suspend:

```
interface SendChannel<in E> {
    suspend fun send(element: E)
    fun close(): Boolean
}

interface ReceiveChannel<out E> {
    suspend fun receive(): E
}

interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

The producer can close a channel to indicate that no more elements are coming.

Several types of channels are defined in the library. They differ in how many elements they can internally store and whether the send() call can be suspended or not. For all of the channel types, the receive() call behaves similarly: it receives an element if the channel is not empty; otherwise, it is suspended.

Unlimited channel

An unlimited channel is the closest analog to a queue: producers can send elements to this channel and it will keep growing indefinitely. The send() call will never be suspended. If the program runs out of memory, you'll get an OutOfMemoryException. The difference between an unlimited channel and a queue is that when a consumer tries to receive from an empty channel, it becomes suspended until some new elements are sent.

Unlimited channel

Buffered channel

The size of a buffered channel is constrained by the specified number. Producers can send elements to this channel until the size limit is reached. All of the elements are internally stored. When the channel is full, the next `send` call on it is suspended until more free space becomes available.

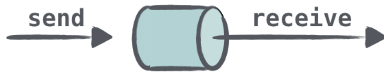
Buffered channel

Rendezvous channel

The "Rendezvous" channel is a channel without a buffer, the same as a buffered channel with zero size. One of the functions (send() or receive()) is always suspended until the other is called.

If the send() function is called and there's no suspended receive call ready to process the element, then send() is suspended. Similarly, if the receive function is called and the channel is empty or, in other words, there's no suspended send() call ready to send the element, the receive() call is suspended.

The "rendezvous" name ("a meeting at an agreed time and place") refers to the fact that send() and receive() should "meet on time".



Rendezvous channel

Conflated channel

A new element sent to the conflated channel will overwrite the previously sent element, so the receiver will always get only the latest element. The send() call is never suspended.

Conflated channel

When you create a channel, specify its type or the buffer size (if you need a buffered one):

```
val rendezvousChannel = Channel<String>()
val bufferedChannel = Channel<String>(10)
val conflatedChannel = Channel<String>(CONFLATED)
val unlimitedChannel = Channel<String>(UNLIMITED)
```

By default, a "Rendezvous" channel is created.

In the following task, you'll create a "Rendezvous" channel, two producer coroutines, and a consumer coroutine:

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val channel = Channel<String>()
    launch {
        channel.send("A1")
        channel.send("A2")
        log("A done")
    }
    launch {
        channel.send("B1")
        log("B done")
    }
    launch {
        repeat(3) {
            val x = channel.receive()
            log(x)
        }
    }
}

fun log(message: Any?) {
    println("[${Thread.currentThread().name}] $message")
}
```

> Watch this video for a better understanding of channels.

Task 7

In src/tasks/Request7Channels.kt, implement the function loadContributorsChannels() that requests all of the GitHub contributors concurrently and shows intermediate progress at the same time.

Use the previous functions, loadContributorsConcurrent() from Request5Concurrent.kt and loadContributorsProgress() from Request6Progress.kt.

Tip for task 7

Different coroutines that concurrently receive contributor lists for different repositories can send all of the received results to the same channel:

```
val channel = Channel<List<User>>()
for (repo in repos) {
    launch {
        val users = TODO()
        // ...
        channel.send(users)
    }
}
```

Then the elements from this channel can be received one by one and processed:

```
repeat(repos.size) {
    val users = channel.receive()
    // ...
}
```

Since the receive() calls are sequential, no additional synchronization is needed.

Solution for task 7

As with the loadContributorsProgress() function, you can create an allUsers variable to store the intermediate states of the "all contributors" list. Each new list received from the channel is added to the list of all users. You aggregate the result and update the state using the updateResults callback:

```
suspend fun loadContributorsChannels(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) = coroutineScope {

    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    val channel = Channel<List<User>>()
    for (repo in repos) {
        launch {
            val users = service.getRepoContributors(req.org, repo.name)
                .also { logUsers(repo, it) }
                .bodyList()
            channel.send(users)
        }
    }
    var allUsers = emptyList<User>()
    repeat(repos.size) {
        val users = channel.receive()
        allUsers = (allUsers + users).aggregate()
        updateResults(allUsers, it == repos.lastIndex)
    }
}
```

- Results for different repositories are added to the channel as soon as they are ready. At first, when all of the requests are sent, and no data is received, the receive() call is suspended. In this case, the whole "load contributors" coroutine is suspended.
- Then, when the list of users is sent to the channel, the "load contributors" coroutine resumes, the receive() call returns this list, and the results are immediately updated.

You can now run the program and choose the CHANNELS option to load the contributors and see the result.

Although neither coroutines nor channels completely remove the complexity that comes with concurrency, they make life easier when you need to understand what's going on.

Testing coroutines

Let's now test all solutions to check that the solution with concurrent coroutines is faster than the solution with the suspend functions, and check that the solution with channels is faster than the simple "progress" one.

In the following task, you'll compare the total running time of the solutions. You'll mock a GitHub service and make this service return results after the given timeouts:

```
repos request - returns an answer within 1000 ms delay
repo-1 - 1000 ms delay
repo-2 - 1200 ms delay
repo-3 - 800 ms delay
```

The sequential solution with the suspend functions should take around 4000 ms (4000 = 1000 + (1000 + 1200 + 800)). The concurrent solution should take around 2200 ms (2200 = 1000 + max(1000, 1200, 800)).

For the solutions that show progress, you can also check the intermediate results with timestamps.

The corresponding test data is defined in test/contributors/testData.kt, and the files Request4SuspendKtTest, Request7ChannelsKtTest, and so on contain the straightforward tests that use mock service calls.

However, there are two problems here:

- These tests take too long to run. Each test takes around 2 to 4 seconds, and you need to wait for the results each time. It's not very efficient.
- You can't rely on the exact time the solution runs because it still takes additional time to prepare and run the code. You could add a constant, but then the time would differ from machine to machine. The mock service delays should be higher than this constant so you can see a difference. If the constant is 0.5 sec, making the delays 0.1 sec won't be enough.

A better way would be to use special frameworks to test the timing while running the same code several times (which increases the total time even more), but that is complicated to learn and set up.

To solve these problems and make sure that solutions with provided test delays behave as expected, one faster than the other, use virtual time with a special test dispatcher. This dispatcher keeps track of the virtual time passed from the start and runs everything immediately in real time. When you run coroutines on this dispatcher, the delay will return immediately and advance the virtual time.

Tests that use this mechanism run fast, but you can still check what happens at different moments in virtual time. The total running time drastically decreases:

Real time:		Virtual time:	
tasks.Request4SuspendKtTest	4 s 16 ms	tasks.Request4SuspendKtTest	3 ms
tasks.Request5ConcurrentKtTest	2 s 214 ms	tasks.Request5ConcurrentKtTest	6 ms
tasks.Request6ProgressKtTest	4 s 16 ms	tasks.Request6ProgressKtTest	3 ms
tasks.Request7ChannelsKtTest	2 s 276 ms	tasks.Request7ChannelsKtTest	3 ms

Comparison for total running time

To use virtual time, replace the runBlocking invocation with a runTest. runTest takes an extension lambda to TestScope as an argument. When you call delay in a suspend function inside this special scope, delay will increase the virtual time instead of delaying in real time:

```
@Test
fun testDelayInSuspend() = runTest {
    val realStartTime = System.currentTimeMillis()
    val virtualStartTime = currentTime

    foo()
    println("${System.currentTimeMillis() - realStartTime} ms") // ~ 6 ms
    println("${currentTime - virtualStartTime} ms")             // 1000 ms
}

suspend fun foo() {
    delay(1000)    // auto-advances without delay
    println("foo") // executes eagerly when foo() is called
}
```

You can check the current virtual time using the currentTime property of TestScope.

The actual running time in this example is several milliseconds, whereas virtual time equals the delay argument, which is 1000 milliseconds.

To get the full effect of "virtual" delay in child coroutines, start all of the child coroutines with TestDispatcher. Otherwise, it won't work. This dispatcher is automatically inherited from the other TestScope, unless you provide a different dispatcher:

```
@Test
fun testDelayInLaunch() = runTest {
    val realStartTime = System.currentTimeMillis()
    val virtualStartTime = currentTime

    bar()

    println("${System.currentTimeMillis() - realStartTime} ms") // ~ 11 ms
    println("${currentTime - virtualStartTime} ms")             // 1000 ms
}

suspend fun bar() = coroutineScope {
    launch {
        delay(1000)    // auto-advances without delay
```

```
        println("bar") // executes eagerly when bar() is called
    }
}
```

If launch is called with the context of Dispatchers.Default in the example above, the test will fail. You'll get an exception saying that the job has not been completed yet.

You can test the loadContributorsConcurrent() function this way only if it starts the child coroutines with the inherited context, without modifying it using the Dispatchers.Default dispatcher.

You can specify the context elements like the dispatcher when calling a function rather than when defining it, which allows for more flexibility and easier testing.

> The testing API that supports virtual time is Experimental and may change in the future.

By default, the compiler shows warnings if you use the experimental testing API. To suppress these warnings, annotate the test function or the whole class containing the tests with @OptIn(ExperimentalCoroutinesApi::class). Add the compiler argument instructing the compiler that you're using the experimental API:

```
compileTestKotlin {
    kotlinOptions {
        freeCompilerArgs += "-Xuse-experimental=kotlin.Experimental"
    }
}
```

In the project corresponding to this tutorial, the compiler argument has already been added to the Gradle script.

Task 8

Refactor the following tests in tests/tasks/ to use virtual time instead of real time:

- Request4SuspendKtTest.kt
- Request5ConcurrentKtTest.kt
- Request6ProgressKtTest.kt
- Request7ChannelsKtTest.kt

Compare the total running times before and after applying your refactoring.

Tip for task 8

1. Replace the runBlocking invocation with runTest, and replace System.currentTimeMillis() with currentTime:

```
@Test
fun test() = runTest {
    val startTime = currentTime
    // action
    val totalTime = currentTime - startTime
    // testing result
}
```

2. Uncomment the assertions that check the exact virtual time.
3. Don't forget to add @UseExperimental(ExperimentalCoroutinesApi::class).

Solution for task 8

Here are the solutions for the concurrent and channels cases:

```
fun testConcurrent() = runTest {
    val startTime = currentTime
    val result = loadContributorsConcurrent(MockGithubService, testRequestData)
    Assert.assertEquals("Wrong result for 'loadContributorsConcurrent'", expectedConcurrentResults.users, result)
    val totalTime = currentTime - startTime

    Assert.assertEquals(
        "The calls run concurrently, so the total virtual time should be 2200 ms: " +
```

```
                "1000 for repos request plus max(1000, 1200, 800) = 1200 for concurrent contributors requests)",
        expectedConcurrentResults.timeFromStart, totalTime
    )
}
```

First, check that the results are available exactly at the expected virtual time, and then check the results themselves:

```
fun testChannels() = runTest {
    val startTime = currentTime
    var index = 0
    loadContributorsChannels(MockGithubService, testRequestData) { users, _ ->
        val expected = concurrentProgressResults[index++]
        val time = currentTime - startTime
        Assert.assertEquals(
            "Expected intermediate results after ${expected.timeFromStart} ms:",
            expected.timeFromStart, time
        )
        Assert.assertEquals("Wrong intermediate results after $time:", expected.users, users)
    }
}
```

The first intermediate result for the last version with channels becomes available sooner than the progress version, and you can see the difference in tests that use virtual time.

> The tests for the remaining "suspend" and "progress" tasks are very similar – you can find them in the project's solutions branch.

What's next

- Check out the Asynchronous Programming with Kotlin workshop at KotlinConf.
- Find out more about using virtual time and the experimental testing package.

Cancellation and timeouts

This section covers coroutine cancellation and timeouts.

Cancelling coroutine execution

In a long-running application you might need fine-grained control on your background coroutines. For example, a user might have closed the page that launched a coroutine and now its result is no longer needed and its operation can be cancelled. The launch function returns a Job that can be used to cancel the running coroutine:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancel() // cancels the job
    job.join() // waits for job's completion
    println("main: Now I can quit.")
//sampleEnd
}
```

> You can get the full code here.

It produces the following output:

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
main: Now I can quit.
```

As soon as main invokes job.cancel, we don't see any output from the other coroutine because it was cancelled. There is also a Job extension function cancelAndJoin that combines cancel and join invocations.

Cancellation is cooperative

Coroutine cancellation is cooperative. A coroutine code has to cooperate to be cancellable. All the suspending functions in kotlinx.coroutines are cancellable. They check for cancellation of coroutine and throw CancellationException when cancelled. However, if a coroutine is working in a computation and does not check for cancellation, then it cannot be cancelled, like the following example shows:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (i < 5) { // computation loop, just wastes CPU
            // print a message twice a second
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}
```

You can get the full code here.

Run it to see that it continues to print "I'm sleeping" even after cancellation until the job completes by itself after five iterations.

The same problem can be observed by catching a CancellationException and not rethrowing it:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch(Dispatchers.Default) {
        repeat(5) { i ->
            try {
                // print a message twice a second
                println("job: I'm sleeping $i ...")
                delay(500)
            } catch (e: Exception) {
                // log the exception
                println(e)
            }
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}
```

You can get the full code here.

While catching Exception is an anti-pattern, this issue may surface in more subtle ways, like when using the runCatching function, which does not rethrow CancellationException.

Making computation code cancellable

There are two approaches to making computation code cancellable. The first one is to periodically invoke a suspending function that checks for cancellation. There is a yield function that is a good choice for that purpose. The other one is to explicitly check the cancellation status. Let us try the latter approach.

Replace while (i < 5) in the previous example with while (isActive) and rerun it.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (isActive) { // cancellable computation loop
            // print a message twice a second
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}
```

You can get the full code here.

As you can see, now this loop is cancelled. isActive is an extension property available inside the coroutine via the CoroutineScope object.

Closing resources with finally

Cancellable suspending functions throw CancellationException on cancellation, which can be handled in the usual way. For example, the try {...} finally {...} expression and Kotlin's use function execute their finalization actions normally when a coroutine is cancelled:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        try {
            repeat(1000) { i ->
                println("job: I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("job: I'm running finally")
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}
```

> You can get the full code here.

Both join and cancelAndJoin wait for all finalization actions to complete, so the example above produces the following output:

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
job: I'm running finally
main: Now I can quit.
```

Run non-cancellable block

Any attempt to use a suspending function in the finally block of the previous example causes CancellationException, because the coroutine running this code is cancelled. Usually, this is not a problem, since all well-behaving closing operations (closing a file, cancelling a job, or closing any kind of a communication channel) are usually non-blocking and do not involve any suspending functions. However, in the rare case when you need to suspend in a cancelled coroutine you can wrap the corresponding code in withContext(NonCancellable) {...} using withContext function and NonCancellable context as the following example shows:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        try {
            repeat(1000) { i ->
                println("job: I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            withContext(NonCancellable) {
                println("job: I'm running finally")
                delay(1000L)
                println("job: And I've just delayed for 1 sec because I'm non-cancellable")
            }
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}
```

> You can get the full code here.

Timeout

The most obvious practical reason to cancel execution of a coroutine is because its execution time has exceeded some timeout. While you can manually track the reference to the corresponding Job and launch a separate coroutine to cancel the tracked one after delay, there is a ready to use withTimeout function that does it. Look at the following example:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    withTimeout(1300L) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
//sampleEnd
}
```

You can get the full code here.

It produces the following output:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Exception in thread "main" kotlinx.coroutines.TimeoutCancellationException: Timed out waiting for 1300 ms
```

The TimeoutCancellationException that is thrown by withTimeout is a subclass of CancellationException. We have not seen its stack trace printed on the console before. That is because inside a cancelled coroutine CancellationException is considered to be a normal reason for coroutine completion. However, in this example we have used withTimeout right inside the main function.

Since cancellation is just an exception, all resources are closed in the usual way. You can wrap the code with timeout in a try {...} catch (e: TimeoutCancellationException) {...} block if you need to do some additional action specifically on any kind of timeout or use the withTimeoutOrNull function that is similar to withTimeout but returns null on timeout instead of throwing an exception:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val result = withTimeoutOrNull(1300L) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
        "Done" // will get cancelled before it produces this result
    }
    println("Result is $result")
//sampleEnd
}
```

You can get the full code here.

There is no longer an exception when running this code:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Result is null
```

Asynchronous timeout and resources

The timeout event in withTimeout is asynchronous with respect to the code running in its block and may happen at any time, even right before the return from inside of the timeout block. Keep this in mind if you open or acquire some resource inside the block that needs closing or release outside of the block.

For example, here we imitate a closeable resource with the Resource class that simply keeps track of how many times it was created by incrementing the acquired counter and decrementing the counter in its close function. Now let us create a lot of coroutines, each of which creates a Resource at the end of the withTimeout block and releases the resource outside the block. We add a small delay so that it is more likely that the timeout occurs right when the withTimeout block is already finished, which will cause a resource leak.

```kotlin
import kotlinx.coroutines.*

//sampleStart
var acquired = 0

class Resource {
    init { acquired++ } // Acquire the resource
    fun close() { acquired-- } // Release the resource
}

fun main() {
    runBlocking {
        repeat(10_000) { // Launch 10K coroutines
```

```
            launch {
                val resource = withTimeout(60) { // Timeout of 60 ms
                    delay(50) // Delay for 50 ms
                    Resource() // Acquire a resource and return it from withTimeout block
                }
                resource.close() // Release the resource
            }
        }
    }
    // Outside of runBlocking all coroutines have completed
    println(acquired) // Print the number of resources still acquired
}
//sampleEnd
```

You can get the full code here.

If you run the above code, you'll see that it does not always print zero, though it may depend on the timings of your machine. You may need to tweak the timeout in this example to actually see non-zero values.

Note that incrementing and decrementing acquired counter here from 10K coroutines is completely thread-safe, since it always happens from the same thread, the one used by runBlocking. More on that will be explained in the chapter on coroutine context.

To work around this problem you can store a reference to the resource in a variable instead of returning it from the withTimeout block.

```
import kotlinx.coroutines.*

var acquired = 0

class Resource {
    init { acquired++ } // Acquire the resource
    fun close() { acquired-- } // Release the resource
}

fun main() {
//sampleStart
    runBlocking {
        repeat(10_000) { // Launch 10K coroutines
            launch {
                var resource: Resource? = null // Not acquired yet
                try {
                    withTimeout(60) { // Timeout of 60 ms
                        delay(50) // Delay for 50 ms
                        resource = Resource() // Store a resource to the variable if acquired
                    }
                    // We can do something else with the resource here
                } finally {
                    resource?.close() // Release the resource if it was acquired
                }
            }
        }
    }
    // Outside of runBlocking all coroutines have completed
    println(acquired) // Print the number of resources still acquired
//sampleEnd
}
```

You can get the full code here.

This example always prints zero. Resources do not leak.

Composing suspending functions

This section covers various approaches to composition of suspending functions.

Sequential by default

Assume that we have two suspending functions defined elsewhere that do something useful like some kind of remote service call or computation. We just pretend they are useful, but actually each one just delays for a second for the purpose of this example:

```
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

What do we do if we need them to be invoked sequentially — first doSomethingUsefulOne and then doSomethingUsefulTwo, and compute the sum of their results? In practice, we do this if we use the result of the first function to make a decision on whether we need to invoke the second one or to decide on how to invoke it.

We use a normal sequential invocation, because the code in the coroutine, just like in the regular code, is sequential by default. The following example demonstrates it by measuring the total time it takes to execute both suspending functions:

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        val one = doSomethingUsefulOne()
        val two = doSomethingUsefulTwo()
        println("The answer is ${one + two}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

You can get the full code here.

It produces something like this:

```
The answer is 42
Completed in 2017 ms
```

Concurrent using async

What if there are no dependencies between invocations of doSomethingUsefulOne and doSomethingUsefulTwo and we want to get the answer faster, by doing both concurrently? This is where async comes to help.

Conceptually, async is just like launch. It starts a separate coroutine which is a light-weight thread that works concurrently with all the other coroutines. The difference is that launch returns a Job and does not carry any resulting value, while async returns a Deferred — a light-weight non-blocking future that represents a promise to provide a result later. You can use .await() on a deferred value to get its eventual result, but Deferred is also a Job, so you can cancel it if needed.

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
```

```
        val one = async { doSomethingUsefulOne() }
        val two = async { doSomethingUsefulTwo() }
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

> You can get the full code here.

It produces something like this:

```
The answer is 42
Completed in 1017 ms
```

This is twice as fast, because the two coroutines execute concurrently. Note that concurrency with coroutines is always explicit.

Lazily started async

Optionally, async can be made lazy by setting its start parameter to CoroutineStart.LAZY. In this mode it only starts the coroutine when its result is required by await, or if its Job's start function is invoked. Run the following example:

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }
        val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }
        // some computation
        one.start() // start the first one
        two.start() // start the second one
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

> You can get the full code here.

It produces something like this:

```
The answer is 42
Completed in 1017 ms
```

So, here the two coroutines are defined but not executed as in the previous example, but the control is given to the programmer on when exactly to start the execution by calling start. We first start one, then start two, and then await for the individual coroutines to finish.

Note that if we just call await in println without first calling start on individual coroutines, this will lead to sequential behavior, since await starts the coroutine execution and waits for its finish, which is not the intended use-case for laziness. The use-case for async(start = CoroutineStart.LAZY) is a replacement for the standard lazy function in cases when computation of the value involves suspending functions.

Async-style functions

We can define async-style functions that invoke doSomethingUsefulOne and doSomethingUsefulTwo asynchronously using the async coroutine builder using a GlobalScope reference to opt-out of the structured concurrency. We name such functions with the "...Async" suffix to highlight the fact that they only start asynchronous computation and one needs to use the resulting deferred value to get the result.

> GlobalScope is a delicate API that can backfire in non-trivial ways, one of which will be explained below, so you must explicitly opt-in into using GlobalScope with @OptIn(DelicateCoroutinesApi::class).

```
// The result type of somethingUsefulOneAsync is Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

// The result type of somethingUsefulTwoAsync is Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}
```

Note that these xxxAsync functions are not suspending functions. They can be used from anywhere. However, their use always implies asynchronous (here meaning concurrent) execution of their action with the invoking code.

The following example shows their use outside of coroutine:

```
import kotlinx.coroutines.*
import kotlin.system.*

//sampleStart
// note that we don't have `runBlocking` to the right of `main` in this example
fun main() {
    val time = measureTimeMillis {
        // we can initiate async actions outside of a coroutine
        val one = somethingUsefulOneAsync()
        val two = somethingUsefulTwoAsync()
        // but waiting for a result must involve either suspending or blocking.
        // here we use `runBlocking { ... }` to block the main thread while waiting for the result
        runBlocking {
            println("The answer is ${one.await() + two.await()}")
        }
    }
    println("Completed in $time ms")
}
//sampleEnd

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
```

```
    return 29
}
```

You can get the full code here.

This programming style with async functions is provided here only for illustration, because it is a popular style in other programming languages. Using this style with Kotlin coroutines is strongly discouraged for the reasons explained below.

Consider what happens if between the val one = somethingUsefulOneAsync() line and one.await() expression there is some logic error in the code, and the program throws an exception, and the operation that was being performed by the program aborts. Normally, a global error-handler could catch this exception, log and report the error for developers, but the program could otherwise continue doing other operations. However, here we have somethingUsefulOneAsync still running in the background, even though the operation that initiated it was aborted. This problem does not happen with structured concurrency, as shown in the section below.

Structured concurrency with async

Let us take the Concurrent using async example and extract a function that concurrently performs doSomethingUsefulOne and doSomethingUsefulTwo and returns the sum of their results. Because the async coroutine builder is defined as an extension on CoroutineScope, we need to have it in the scope and that is what the coroutineScope function provides:

```
suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}
```

This way, if something goes wrong inside the code of the concurrentSum function, and it throws an exception, all the coroutines that were launched in its scope will be cancelled.

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        println("The answer is ${concurrentSum()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

You can get the full code here.

We still have concurrent execution of both operations, as evident from the output of the above main function:

```
The answer is 42
Completed in 1017 ms
```

Cancellation is always propagated through coroutines hierarchy:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    try {
        failedConcurrentSum()
    } catch(e: ArithmeticException) {
        println("Computation failed with ArithmeticException")
    }
}

suspend fun failedConcurrentSum(): Int = coroutineScope {
    val one = async<Int> {
        try {
            delay(Long.MAX_VALUE) // Emulates very long computation
            42
        } finally {
            println("First child was cancelled")
        }
    }
    val two = async<Int> {
        println("Second child throws an exception")
        throw ArithmeticException()
    }
    one.await() + two.await()
}
```

You can get the full code here.

Note how both the first async and the awaiting parent are cancelled on failure of one of the children (namely, two):

```
Second child throws an exception
First child was cancelled
Computation failed with ArithmeticException
```

Coroutine context and dispatchers

Coroutines always execute in some context represented by a value of the CoroutineContext type, defined in the Kotlin standard library.

The coroutine context is a set of various elements. The main elements are the Job of the coroutine, which we've seen before, and its dispatcher, which is covered in this section.

Dispatchers and threads

The coroutine context includes a coroutine dispatcher (see CoroutineDispatcher) that determines what thread or threads the corresponding coroutine uses for its execution. The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.

All coroutine builders like launch and async accept an optional CoroutineContext parameter that can be used to explicitly specify the dispatcher for the new coroutine and other context elements.

Try the following example:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch { // context of the parent, main runBlocking coroutine
        println("main runBlocking      : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
        println("Unconfined            : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
        println("Default               : I'm working in thread ${Thread.currentThread().name}")
    }
```

```
    launch(newSingleThreadContext("MyOwnThread")) { // will get its own new thread
        println("newSingleThreadContext: I'm working in thread ${Thread.currentThread().name}")
    }
//sampleEnd
}
```

> You can get the full code here.

It produces the following output (maybe in different order):

```
Unconfined            : I'm working in thread main
Default               : I'm working in thread DefaultDispatcher-worker-1
newSingleThreadContext: I'm working in thread MyOwnThread
main runBlocking      : I'm working in thread main
```

When launch { ... } is used without parameters, it inherits the context (and thus dispatcher) from the CoroutineScope it is being launched from. In this case, it inherits the context of the main runBlocking coroutine which runs in the main thread.

Dispatchers.Unconfined is a special dispatcher that also appears to run in the main thread, but it is, in fact, a different mechanism that is explained later.

The default dispatcher is used when no other dispatcher is explicitly specified in the scope. It is represented by Dispatchers.Default and uses a shared background pool of threads.

newSingleThreadContext creates a thread for the coroutine to run. A dedicated thread is a very expensive resource. In a real application it must be either released, when no longer needed, using the close function, or stored in a top-level variable and reused throughout the application.

Unconfined vs confined dispatcher

The Dispatchers.Unconfined coroutine dispatcher starts a coroutine in the caller thread, but only until the first suspension point. After suspension it resumes the coroutine in the thread that is fully determined by the suspending function that was invoked. The unconfined dispatcher is appropriate for coroutines which neither consume CPU time nor update any shared data (like UI) confined to a specific thread.

On the other side, the dispatcher is inherited from the outer CoroutineScope by default. The default dispatcher for the runBlocking coroutine, in particular, is confined to the invoker thread, so inheriting it has the effect of confining execution to this thread with predictable FIFO scheduling.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
        println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")
        delay(500)
        println("Unconfined      : After delay in thread ${Thread.currentThread().name}")
    }
    launch { // context of the parent, main runBlocking coroutine
        println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")
        delay(1000)
        println("main runBlocking: After delay in thread ${Thread.currentThread().name}")
    }
//sampleEnd
}
```

> You can get the full code here.

Produces the output:

```
Unconfined      : I'm working in thread main
main runBlocking: I'm working in thread main
Unconfined      : After delay in thread kotlinx.coroutines.DefaultExecutor
main runBlocking: After delay in thread main
```

So, the coroutine with the context inherited from runBlocking {...} continues to execute in the main thread, while the unconfined one resumes in the default executor thread that the delay function is using.

The unconfined dispatcher is an advanced mechanism that can be helpful in certain corner cases where dispatching of a coroutine for its execution later is not needed or produces undesirable side-effects, because some operation in a coroutine must be performed right away. The unconfined dispatcher should not be used in general code.

Debugging coroutines and threads

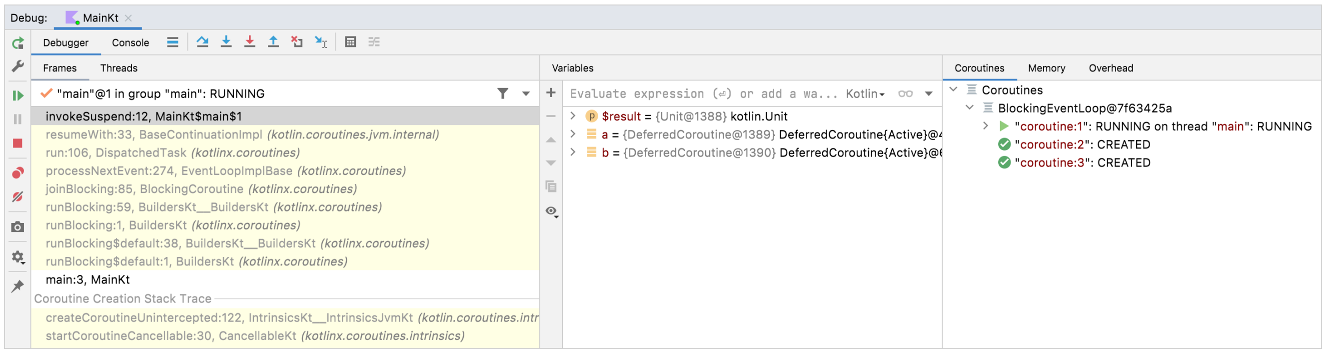
Coroutines can suspend on one thread and resume on another thread. Even with a single-threaded dispatcher it might be hard to figure out what the coroutine was doing, where, and when if you don't have special tooling.

Debugging with IDEA

The Coroutine Debugger of the Kotlin plugin simplifies debugging coroutines in IntelliJ IDEA.

Debugging works for versions 1.3.8 or later of kotlinx-coroutines-core.

The Debug tool window contains the Coroutines tab. In this tab, you can find information about both currently running and suspended coroutines. The coroutines are grouped by the dispatcher they are running on.



Debugging coroutines

With the coroutine debugger, you can:

- Check the state of each coroutine.
- See the values of local and captured variables for both running and suspended coroutines.
- See a full coroutine creation stack, as well as a call stack inside the coroutine. The stack includes all frames with variable values, even those that would be lost during standard debugging.
- Get a full report that contains the state of each coroutine and its stack. To obtain it, right-click inside the Coroutines tab, and then click Get Coroutines Dump.

To start coroutine debugging, you just need to set breakpoints and run the application in debug mode.

Learn more about coroutines debugging in the tutorial.

Debugging using logging

Another approach to debugging applications with threads without Coroutine Debugger is to print the thread name in the log file on each log statement. This feature is universally supported by logging frameworks. When using coroutines, the thread name alone does not give much of a context, so kotlinx.coroutines includes debugging facilities to make it easier.

Run the following code with -Dkotlinx.coroutines.debug JVM option:

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking<Unit> {
```

```
//sampleStart
    val a = async {
        log("I'm computing a piece of the answer")
        6
    }
    val b = async {
        log("I'm computing another piece of the answer")
        7
    }
    log("The answer is ${a.await() * b.await()}")
//sampleEnd
}
```

You can get the full code here.

There are three coroutines. The main coroutine (#1) inside runBlocking and two coroutines computing the deferred values a (#2) and b (#3). They are all executing in the context of runBlocking and are confined to the main thread. The output of this code is:

```
[main @coroutine#2] I'm computing a piece of the answer
[main @coroutine#3] I'm computing another piece of the answer
[main @coroutine#1] The answer is 42
```

The log function prints the name of the thread in square brackets, and you can see that it is the main thread with the identifier of the currently executing coroutine appended to it. This identifier is consecutively assigned to all created coroutines when the debugging mode is on.

Debugging mode is also turned on when JVM is run with -ea option. You can read more about debugging facilities in the documentation of the DEBUG_PROPERTY_NAME property.

Jumping between threads

Run the following code with the -Dkotlinx.coroutines.debug JVM option (see debug):

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() {
//sampleStart
    newSingleThreadContext("Ctx1").use { ctx1 ->
        newSingleThreadContext("Ctx2").use { ctx2 ->
            runBlocking(ctx1) {
                log("Started in ctx1")
                withContext(ctx2) {
                    log("Working in ctx2")
                }
                log("Back to ctx1")
            }
        }
    }
//sampleEnd
}
```

You can get the full code here.

It demonstrates several new techniques. One is using runBlocking with an explicitly specified context, and the other one is using the withContext function to change the context of a coroutine while still staying in the same coroutine, as you can see in the output below:

```
[Ctx1 @coroutine#1] Started in ctx1
[Ctx2 @coroutine#1] Working in ctx2
[Ctx1 @coroutine#1] Back to ctx1
```

Note that this example also uses the use function from the Kotlin standard library to release threads created with newSingleThreadContext when they are no longer

needed.

Job in the context

The coroutine's Job is part of its context, and can be retrieved from it using the coroutineContext[Job] expression:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    println("My job is ${coroutineContext[Job]}")
//sampleEnd
}
```

> You can get the full code here.

In the debug mode, it outputs something like this:

My job is "coroutine#1":BlockingCoroutine{Active}@6d311334

Note that isActive in CoroutineScope is just a convenient shortcut for coroutineContext[Job]?.isActive == true.

Children of a coroutine

When a coroutine is launched in the CoroutineScope of another coroutine, it inherits its context via CoroutineScope.coroutineContext and the Job of the new coroutine becomes a child of the parent coroutine's job. When the parent coroutine is cancelled, all its children are recursively cancelled, too.

However, this parent-child relation can be explicitly overriden in one of two ways:

1. When a different scope is explicitly specified when launching a coroutine (for example, GlobalScope.launch), then it does not inherit a Job from the parent scope.
2. When a different Job object is passed as the context for the new coroutine (as shown in the example below), then it overrides the Job of the parent scope.

In both cases, the launched coroutine is not tied to the scope it was launched from and operates independently.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    // launch a coroutine to process some kind of incoming request
    val request = launch {
        // it spawns two other jobs
        launch(Job()) {
            println("job1: I run in my own Job and execute independently!")
            delay(1000)
            println("job1: I am not affected by cancellation of the request")
        }
        // and the other inherits the parent context
        launch {
            delay(100)
            println("job2: I am a child of the request coroutine")
            delay(1000)
            println("job2: I will not execute this line if my parent request is cancelled")
        }
    }
    delay(500)
    request.cancel() // cancel processing of the request
    println("main: Who has survived request cancellation?")
    delay(1000) // delay the main thread for a second to see what happens
//sampleEnd
}
```

> You can get the full code here.

The output of this code is:

```
job1: I run in my own Job and execute independently!
job2: I am a child of the request coroutine
main: Who has survived request cancellation?
job1: I am not affected by cancellation of the request
```

Parental responsibilities

A parent coroutine always waits for completion of all its children. A parent does not have to explicitly track all the children it launches, and it does not have to use Job.join to wait for them at the end:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    // launch a coroutine to process some kind of incoming request
    val request = launch {
        repeat(3) { i -> // launch a few children jobs
            launch  {
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, 600ms
                println("Coroutine $i is done")
            }
        }
        println("request: I'm done and I don't explicitly join my children that are still active")
    }
    request.join() // wait for completion of the request, including all its children
    println("Now processing of the request is complete")
//sampleEnd
}
```

You can get the full code here.

The result is going to be:

```
request: I'm done and I don't explicitly join my children that are still active
Coroutine 0 is done
Coroutine 1 is done
Coroutine 2 is done
Now processing of the request is complete
```

Naming coroutines for debugging

Automatically assigned ids are good when coroutines log often and you just need to correlate log records coming from the same coroutine. However, when a coroutine is tied to the processing of a specific request or doing some specific background task, it is better to name it explicitly for debugging purposes. The CoroutineName context element serves the same purpose as the thread name. It is included in the thread name that is executing this coroutine when the debugging mode is turned on.

The following example demonstrates this concept:

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking(CoroutineName("main")) {
//sampleStart
    log("Started main coroutine")
    // run two background value computations
    val v1 = async(CoroutineName("v1coroutine")) {
        delay(500)
        log("Computing v1")
        252
    }
    val v2 = async(CoroutineName("v2coroutine")) {
        delay(1000)
        log("Computing v2")
```

```
        6
    }
    log("The answer for v1 / v2 = ${v1.await() / v2.await()}")
//sampleEnd
}
```

> You can get the full code here.

The output it produces with -Dkotlinx.coroutines.debug JVM option is similar to:

```
[main @main#1] Started main coroutine
[main @v1coroutine#2] Computing v1
[main @v2coroutine#3] Computing v2
[main @main#1] The answer for v1 / v2 = 42
```

Combining context elements

Sometimes we need to define multiple elements for a coroutine context. We can use the + operator for that. For example, we can launch a coroutine with an explicitly specified dispatcher and an explicitly specified name at the same time:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch(Dispatchers.Default + CoroutineName("test")) {
        println("I'm working in thread ${Thread.currentThread().name}")
    }
//sampleEnd
}
```

> You can get the full code here.

The output of this code with the -Dkotlinx.coroutines.debug JVM option is:

```
I'm working in thread DefaultDispatcher-worker-1 @test#2
```

Coroutine scope

Let us put our knowledge about contexts, children and jobs together. Assume that our application has an object with a lifecycle, but that object is not a coroutine. For example, we are writing an Android application and launch various coroutines in the context of an Android activity to perform asynchronous operations to fetch and update data, do animations, etc. All of these coroutines must be cancelled when the activity is destroyed to avoid memory leaks. We, of course, can manipulate contexts and jobs manually to tie the lifecycles of the activity and its coroutines, but kotlinx.coroutines provides an abstraction encapsulating that: CoroutineScope. You should be already familiar with the coroutine scope as all coroutine builders are declared as extensions on it.

We manage the lifecycles of our coroutines by creating an instance of CoroutineScope tied to the lifecycle of our activity. A CoroutineScope instance can be created by the CoroutineScope() or MainScope() factory functions. The former creates a general-purpose scope, while the latter creates a scope for UI applications and uses Dispatchers.Main as the default dispatcher:

```
class Activity {
    private val mainScope = MainScope()

    fun destroy() {
        mainScope.cancel()
    }
    // to be continued ...
```

Now, we can launch coroutines in the scope of this Activity using the defined scope. For the demo, we launch ten coroutines that delay for a different time:

```
    // class Activity continues
```

```
    fun doSomething() {
        // launch ten coroutines for a demo, each working for a different time
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, ... etc
                println("Coroutine $i is done")
            }
        }
    }
} // class Activity ends
```

In our main function we create the activity, call our test doSomething function, and destroy the activity after 500ms. This cancels all the coroutines that were launched from doSomething. We can see that because after the destruction of the activity no more messages are printed, even if we wait a little longer.

```
import kotlinx.coroutines.*

class Activity {
    private val mainScope = CoroutineScope(Dispatchers.Default) // use Default for test purposes

    fun destroy() {
        mainScope.cancel()
    }

    fun doSomething() {
        // launch ten coroutines for a demo, each working for a different time
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, ... etc
                println("Coroutine $i is done")
            }
        }
    }
} // class Activity ends

fun main() = runBlocking<Unit> {
//sampleStart
    val activity = Activity()
    activity.doSomething() // run test function
    println("Launched coroutines")
    delay(500L) // delay for half a second
    println("Destroying activity!")
    activity.destroy() // cancels all coroutines
    delay(1000) // visually confirm that they don't work
//sampleEnd
}
```

You can get the full code here.

The output of this example is:

```
Launched coroutines
Coroutine 0 is done
Coroutine 1 is done
Destroying activity!
```

As you can see, only the first two coroutines print a message and the others are cancelled by a single invocation of job.cancel() in Activity.destroy().

Note, that Android has first-party support for coroutine scope in all entities with the lifecycle. See the corresponding documentation.

Thread-local data

Sometimes it is convenient to have an ability to pass some thread-local data to or between coroutines. However, since they are not bound to any particular thread, this will likely lead to boilerplate if done manually.

For ThreadLocal, the asContextElement extension function is here for the rescue. It creates an additional context element which keeps the value of the given ThreadLocal and restores it every time the coroutine switches its context.

It is easy to demonstrate it in action:

```kotlin
import kotlinx.coroutines.*

val threadLocal = ThreadLocal<String?>() // declare thread-local variable

fun main() = runBlocking<Unit> {
//sampleStart
    threadLocal.set("main")
    println("Pre-main, current thread: ${Thread.currentThread()}, thread local value: '${threadLocal.get()}'")
    val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "launch")) {
        println("Launch start, current thread: ${Thread.currentThread()}, thread local value: '${threadLocal.get()}'")
        yield()
        println("After yield, current thread: ${Thread.currentThread()}, thread local value: '${threadLocal.get()}'")
    }
    job.join()
    println("Post-main, current thread: ${Thread.currentThread()}, thread local value: '${threadLocal.get()}'")
//sampleEnd
}
```

You can get the full code here.

In this example we launch a new coroutine in a background thread pool using Dispatchers.Default, so it works on a different thread from the thread pool, but it still has the value of the thread local variable that we specified using threadLocal.asContextElement(value = "launch"), no matter which thread the coroutine is executed on. Thus, the output (with debug) is:

```
Pre-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'
Launch start, current thread: Thread[DefaultDispatcher-worker-1 @coroutine#2,5,main], thread local value: 'launch'
After yield, current thread: Thread[DefaultDispatcher-worker-2 @coroutine#2,5,main], thread local value: 'launch'
Post-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'
```

It's easy to forget to set the corresponding context element. The thread-local variable accessed from the coroutine may then have an unexpected value, if the thread running the coroutine is different. To avoid such situations, it is recommended to use the ensurePresent method and fail-fast on improper usages.

ThreadLocal has first-class support and can be used with any primitive kotlinx.coroutines provides. It has one key limitation, though: when a thread-local is mutated, a new value is not propagated to the coroutine caller (because a context element cannot track all ThreadLocal object accesses), and the updated value is lost on the next suspension. Use withContext to update the value of the thread-local in a coroutine, see asContextElement for more details.

Alternatively, a value can be stored in a mutable box like class Counter(var i: Int), which is, in turn, stored in a thread-local variable. However, in this case you are fully responsible to synchronize potentially concurrent modifications to the variable in this mutable box.

For advanced usage, for example for integration with logging MDC, transactional contexts or any other libraries which internally use thread-locals for passing data, see the documentation of the ThreadContextElement interface that should be implemented.

Asynchronous Flow

A suspending function asynchronously returns a single value, but how can we return multiple asynchronously computed values? This is where Kotlin Flows come in.

Representing multiple values

Multiple values can be represented in Kotlin using collections. For example, we can have a simple function that returns a List of three numbers and then print them all using forEach:

```kotlin
fun simple(): List<Int> = listOf(1, 2, 3)

fun main() {
    simple().forEach { value -> println(value) }
}
```

You can get the full code from here.

This code outputs:

```
1
2
3
```

Sequences

If we are computing the numbers with some CPU-consuming blocking code (each computation taking 100ms), then we can represent the numbers using a Sequence:

```
fun simple(): Sequence<Int> = sequence { // sequence builder
    for (i in 1..3) {
        Thread.sleep(100) // pretend we are computing it
        yield(i) // yield next value
    }
}

fun main() {
    simple().forEach { value -> println(value) }
}
```

You can get the full code from here.

This code outputs the same numbers, but it waits 100ms before printing each one.

Suspending functions

However, this computation blocks the main thread that is running the code. When these values are computed by asynchronous code we can mark the simple function with a suspend modifier, so that it can perform its work without blocking and return the result as a list:

```
import kotlinx.coroutines.*

//sampleStart
suspend fun simple(): List<Int> {
    delay(1000) // pretend we are doing something asynchronous here
    return listOf(1, 2, 3)
}

fun main() = runBlocking<Unit> {
    simple().forEach { value -> println(value) }
}
//sampleEnd
```

You can get the full code from here.

This code prints the numbers after waiting for a second.

Flows

Using the List<Int> result type, means we can only return all the values at once. To represent the stream of values that are being computed asynchronously, we can use a Flow<Int> type just like we would use a Sequence<Int> type for synchronously computed values:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    // Launch a concurrent coroutine to check if the main thread is blocked
```

```
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // Collect the flow
    simple().collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from here.

This code waits 100ms before printing each number without blocking the main thread. This is verified by printing "I'm not blocked" every 100ms from a separate coroutine that is running in the main thread:

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

Notice the following differences in the code with the Flow from the earlier examples:

- A builder function of Flow type is called flow.
- Code inside a flow { ... } builder block can suspend.
- The simple function is no longer marked with a suspend modifier.
- Values are emitted from the flow using an emit function.
- Values are collected from the flow using a collect function.

We can replace delay with Thread.sleep in the body of simple's flow { ... } and see that the main thread is blocked in this case.

Flows are cold

Flows are cold streams similar to sequences — the code inside a flow builder does not run until the flow is collected. This becomes clear in the following example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    println("Flow started")
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    println("Calling simple function...")
    val flow = simple()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from here.

Which prints:

```
Calling simple function...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

This is a key reason the simple function (which returns a flow) is not marked with suspend modifier. The simple() call itself returns quickly and does not wait for anything. The flow starts afresh every time it is collected and that is why we see "Flow started" every time we call collect again.

Flow cancellation basics

Flows adhere to the general cooperative cancellation of coroutines. As usual, flow collection can be cancelled when the flow is suspended in a cancellable suspending function (like delay). The following example shows how the flow gets cancelled on a timeout when running in a withTimeoutOrNull block and stops executing its code:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    withTimeoutOrNull(250) { // Timeout after 250ms
        simple().collect { value -> println(value) }
    }
    println("Done")
}
//sampleEnd
```

You can get the full code from here.

Notice how only two numbers get emitted by the flow in the simple function, producing the following output:

```
Emitting 1
1
Emitting 2
2
Done
```

See Flow cancellation checks section for more details.

Flow builders

The flow { ... } builder from the previous examples is the most basic one. There are other builders that allow flows to be declared:

- The flowOf builder defines a flow that emits a fixed set of values.
- Various collections and sequences can be converted to flows using the .asFlow() extension function.

For example, the snippet that prints the numbers 1 to 3 from a flow can be rewritten as follows:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    // Convert an integer range to a flow
    (1..3).asFlow().collect { value -> println(value) }
//sampleEnd
}
```

> You can get the full code from here.

Intermediate flow operators

Flows can be transformed using operators, in the same way as you would transform collections and sequences. Intermediate operators are applied to an upstream flow and return a downstream flow. These operators are cold, just like flows are. A call to such an operator is not a suspending function itself. It works quickly, returning the definition of a new transformed flow.

The basic operators have familiar names like map and filter. An important difference of these operators from sequences is that blocks of code inside these operators can call suspending functions.

For example, a flow of incoming requests can be mapped to its results with a map operator, even when performing a request is a long-running operation that is implemented by a suspending function:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
suspend fun performRequest(request: Int): String {
    delay(1000) // imitate long-running asynchronous work
    return "response $request"
}

fun main() = runBlocking<Unit> {
    (1..3).asFlow() // a flow of requests
        .map { request -> performRequest(request) }
        .collect { response -> println(response) }
}
//sampleEnd
```

> You can get the full code from here.

It produces the following three lines, each appearing one second after the previous:

```
response 1
response 2
response 3
```

Transform operator

Among the flow transformation operators, the most general one is called transform. It can be used to imitate simple transformations like map and filter, as well as implement more complex transformations. Using the transform operator, we can emit arbitrary values an arbitrary number of times.

For example, using transform we can emit a string before performing a long-running asynchronous request and follow it with a response:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun performRequest(request: Int): String {
    delay(1000) // imitate long-running asynchronous work
    return "response $request"
}

fun main() = runBlocking<Unit> {
```

```kotlin
//sampleStart
    (1..3).asFlow() // a flow of requests
        .transform { request ->
            emit("Making request $request")
            emit(performRequest(request))
        }
        .collect { response -> println(response) }
//sampleEnd
}
```

You can get the full code from here.

The output of this code is:

```
Making request 1
response 1
Making request 2
response 2
Making request 3
response 3
```

Size-limiting operators

Size-limiting intermediate operators like take cancel the execution of the flow when the corresponding limit is reached. Cancellation in coroutines is always performed by throwing an exception, so that all the resource-management functions (like try { ... } finally { ... } blocks) operate normally in case of cancellation:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun numbers(): Flow<Int> = flow {
    try {
        emit(1)
        emit(2)
        println("This line will not execute")
        emit(3)
    } finally {
        println("Finally in numbers")
    }
}

fun main() = runBlocking<Unit> {
    numbers()
        .take(2) // take only the first two
        .collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from here.

The output of this code clearly shows that the execution of the flow { ... } body in the numbers() function stopped after emitting the second number:

```
1
2
Finally in numbers
```

Terminal flow operators

Terminal operators on flows are suspending functions that start a collection of the flow. The collect operator is the most basic one, but there are other terminal operators, which can make it easier:

- Conversion to various collections like toList and toSet.
- Operators to get the first value and to ensure that a flow emits a single value.

- Reducing a flow to a value with reduce and fold.

For example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val sum = (1..5).asFlow()
        .map { it * it } // squares of numbers from 1 to 5
        .reduce { a, b -> a + b } // sum them (terminal operator)
    println(sum)
//sampleEnd
}
```

> You can get the full code from here.

Prints a single number:

```
55
```

Flows are sequential

Each individual collection of a flow is performed sequentially unless special operators that operate on multiple flows are used. The collection works directly in the coroutine that calls a terminal operator. No new coroutines are launched by default. Each emitted value is processed by all the intermediate operators from upstream to downstream and is then delivered to the terminal operator after.

See the following example that filters the even integers and maps them to strings:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    (1..5).asFlow()
        .filter {
            println("Filter $it")
            it % 2 == 0
        }
        .map {
            println("Map $it")
            "string $it"
        }.collect {
            println("Collect $it")
        }
//sampleEnd
}
```

> You can get the full code from here.

Producing:

```
Filter 1
Filter 2
Map 2
Collect string 2
Filter 3
Filter 4
Map 4
Collect string 4
Filter 5
```

Flow context

Collection of a flow always happens in the context of the calling coroutine. For example, if there is a simple flow, then the following code runs in the context specified by the author of this code, regardless of the implementation details of the simple flow:

```
withContext(context) {
    simple().collect { value ->
        println(value) // run in the specified context
    }
}
```

This property of a flow is called context preservation.

So, by default, code in the flow { ... } builder runs in the context that is provided by a collector of the corresponding flow. For example, consider the implementation of a simple function that prints the thread it is called on and emits three numbers:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

//sampleStart
fun simple(): Flow<Int> = flow {
    log("Started simple flow")
    for (i in 1..3) {
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> log("Collected $value") }
}
//sampleEnd
```

You can get the full code from here.

Running this code produces:

```
[main @coroutine#1] Started simple flow
[main @coroutine#1] Collected 1
[main @coroutine#1] Collected 2
[main @coroutine#1] Collected 3
```

Since simple().collect is called from the main thread, the body of simple's flow is also called in the main thread. This is the perfect default for fast-running or asynchronous code that does not care about the execution context and does not block the caller.

A common pitfall when using withContext

However, the long-running CPU-consuming code might need to be executed in the context of Dispatchers.Default and UI-updating code might need to be executed in the context of Dispatchers.Main. Usually, withContext is used to change the context in the code using Kotlin coroutines, but code in the flow { ... } builder has to honor the context preservation property and is not allowed to emit from a different context.

Try running the following code:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    // The WRONG way to change context for CPU-consuming code in flow builder
    kotlinx.coroutines.withContext(Dispatchers.Default) {
        for (i in 1..3) {
            Thread.sleep(100) // pretend we are computing it in CPU-consuming way
            emit(i) // emit next value
        }
    }
}
```

```
fun main() = runBlocking<Unit> {
    simple().collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from here.

This code produces the following exception:

```
Exception in thread "main" java.lang.IllegalStateException: Flow invariant is violated:
  Flow was collected in [CoroutineId(1), "coroutine#1":BlockingCoroutine{Active}@5511c7f8, BlockingEventLoop@2eac3323],
  but emission happened in [CoroutineId(1), "coroutine#1":DispatchedCoroutine{Active}@2dae0000, Dispatchers.Default].
  Please refer to 'flow' documentation or use 'flowOn' instead
 at ...
```

flowOn operator

The exception refers to the flowOn function that shall be used to change the context of the flow emission. The correct way to change the context of a flow is shown in the example below, which also prints the names of the corresponding threads to show how it all works:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        Thread.sleep(100) // pretend we are computing it in CPU-consuming way
        log("Emitting $i")
        emit(i) // emit next value
    }
}.flowOn(Dispatchers.Default) // RIGHT way to change context for CPU-consuming code in flow builder

fun main() = runBlocking<Unit> {
    simple().collect { value ->
        log("Collected $value")
    }
}
//sampleEnd
```

You can get the full code from here.

Notice how flow { ... } works in the background thread, while collection happens in the main thread:

Another thing to observe here is that the flowOn operator has changed the default sequential nature of the flow. Now collection happens in one coroutine ("coroutine#1") and emission happens in another coroutine ("coroutine#2") that is running in another thread concurrently with the collecting coroutine. The flowOn operator creates another coroutine for an upstream flow when it has to change the CoroutineDispatcher in its context.

Buffering

Running different parts of a flow in different coroutines can be helpful from the standpoint of the overall time it takes to collect the flow, especially when long-running asynchronous operations are involved. For example, consider a case when the emission by a simple flow is slow, taking 100 ms to produce an element; and collector is also slow, taking 300 ms to process an element. Let's see how long it takes to collect such a flow with three numbers:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // pretend we are asynchronously waiting 100 ms
        emit(i) // emit next value
```

```
    }
}

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        simple().collect { value ->
            delay(300) // pretend we are processing it for 300 ms
            println(value)
        }
    }
    println("Collected in $time ms")
}
//sampleEnd
```

> You can get the full code from here.

It produces something like this, with the whole collection taking around 1200 ms (three numbers, 400 ms for each):

```
1
2
3
Collected in 1220 ms
```

We can use a buffer operator on a flow to run emitting code of the simple flow concurrently with collecting code, as opposed to running them sequentially:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // pretend we are asynchronously waiting 100 ms
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .buffer() // buffer emissions, don't wait
            .collect { value ->
                delay(300) // pretend we are processing it for 300 ms
                println(value)
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

> You can get the full code from here.

It produces the same numbers just faster, as we have effectively created a processing pipeline, having to only wait 100 ms for the first number and then spending only 300 ms to process each number. This way it takes around 1000 ms to run:

```
1
2
3
Collected in 1071 ms
```

> Note that the flowOn operator uses the same buffering mechanism when it has to change a CoroutineDispatcher, but here we explicitly request buffering without changing the execution context.

Conflation

When a flow represents partial results of the operation or operation status updates, it may not be necessary to process each value, but instead, only most recent ones. In this case, the conflate operator can be used to skip intermediate values when a collector is too slow to process them. Building on the previous example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // pretend we are asynchronously waiting 100 ms
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .conflate() // conflate emissions, don't process each one
            .collect { value ->
                delay(300) // pretend we are processing it for 300 ms
                println(value)
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

You can get the full code from here.

We see that while the first number was still being processed the second, and third were already produced, so the second one was conflated and only the most recent (the third one) was delivered to the collector:

```
1
3
Collected in 758 ms
```

Processing the latest value

Conflation is one way to speed up processing when both the emitter and collector are slow. It does it by dropping emitted values. The other way is to cancel a slow collector and restart it every time a new value is emitted. There is a family of xxxLatest operators that perform the same essential logic of a xxx operator, but cancel the code in their block on a new value. Let's try changing conflate to collectLatest in the previous example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // pretend we are asynchronously waiting 100 ms
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .collectLatest { value -> // cancel & restart on the latest value
                println("Collecting $value")
                delay(300) // pretend we are processing it for 300 ms
                println("Done $value")
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

> You can get the full code from here.

Since the body of collectLatest takes 300 ms, but new values are emitted every 100 ms, we see that the block is run on every value, but completes only for the last value:

```
Collecting 1
Collecting 2
Collecting 3
Done 3
Collected in 741 ms
```

Composing multiple flows

There are lots of ways to compose multiple flows.

Zip

Just like the Sequence.zip extension function in the Kotlin standard library, flows have a zip operator that combines the corresponding values of two flows:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow() // numbers 1..3
    val strs = flowOf("one", "two", "three") // strings
    nums.zip(strs) { a, b -> "$a -> $b" } // compose a single string
        .collect { println(it) } // collect and print
//sampleEnd
}
```

> You can get the full code from here.

This example prints:

```
1 -> one
2 -> two
3 -> three
```

Combine

When flow represents the most recent value of a variable or operation (see also the related section on conflation), it might be needed to perform a computation that depends on the most recent values of the corresponding flows and to recompute it whenever any of the upstream flows emit a value. The corresponding family of operators is called combine.

For example, if the numbers in the previous example update every 300ms, but strings update every 400 ms, then zipping them using the zip operator will still produce the same result, albeit results that are printed every 400 ms:

> We use a onEach intermediate operator in this example to delay each element and make the code that emits sample flows more declarative and shorter.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow().onEach { delay(300) } // numbers 1..3 every 300 ms
    val strs = flowOf("one", "two", "three").onEach { delay(400) } // strings every 400 ms
    val startTime = System.currentTimeMillis() // remember the start time
    nums.zip(strs) { a, b -> "$a -> $b" } // compose a single string with "zip"
        .collect { value -> // collect and print
```

```
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
//sampleEnd
}
```

You can get the full code from here.

However, when using a combine operator here instead of a zip:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow().onEach { delay(300) } // numbers 1..3 every 300 ms
    val strs = flowOf("one", "two", "three").onEach { delay(400) } // strings every 400 ms
    val startTime = System.currentTimeMillis() // remember the start time
    nums.combine(strs) { a, b -> "$a -> $b" } // compose a single string with "combine"
        .collect { value -> // collect and print
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
//sampleEnd
}
```

You can get the full code from here.

We get quite a different output, where a line is printed at each emission from either nums or strs flows:

```
1 -> one at 452 ms from start
2 -> one at 651 ms from start
2 -> two at 854 ms from start
3 -> two at 952 ms from start
3 -> three at 1256 ms from start
```

Flattening flows

Flows represent asynchronously received sequences of values, and so it is quite easy to get into a situation where each value triggers a request for another sequence of values. For example, we can have the following function that returns a flow of two strings 500 ms apart:

```
fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // wait 500 ms
    emit("$i: Second")
}
```

Now if we have a flow of three integers and call requestFlow on each of them like this:

```
(1..3).asFlow().map { requestFlow(it) }
```

Then we will end up with a flow of flows (Flow<Flow<String>>) that needs to be flattened into a single flow for further processing. Collections and sequences have flatten and flatMap operators for this. However, due to the asynchronous nature of flows they call for different modes of flattening, and hence, a family of flattening operators on flows exists.

flatMapConcat

Concatenation of flows of flows is provided by the flatMapConcat and flattenConcat operators. They are the most direct analogues of the corresponding sequence operators. They wait for the inner flow to complete before starting to collect the next one as the following example shows:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```kotlin
fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // wait 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // remember the start time
    (1..3).asFlow().onEach { delay(100) } // emit a number every 100 ms
        .flatMapConcat { requestFlow(it) }
        .collect { value -> // collect and print
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
//sampleEnd
}
```

You can get the full code from here.

The sequential nature of flatMapConcat is clearly seen in the output:

```
1: First at 121 ms from start
1: Second at 622 ms from start
2: First at 727 ms from start
2: Second at 1227 ms from start
3: First at 1328 ms from start
3: Second at 1829 ms from start
```

flatMapMerge

Another flattening operation is to concurrently collect all the incoming flows and merge their values into a single flow so that values are emitted as soon as possible. It is implemented by flatMapMerge and flattenMerge operators. They both accept an optional concurrency parameter that limits the number of concurrent flows that are collected at the same time (it is equal to DEFAULT_CONCURRENCY by default).

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // wait 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // remember the start time
    (1..3).asFlow().onEach { delay(100) } // a number every 100 ms
        .flatMapMerge { requestFlow(it) }
        .collect { value -> // collect and print
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
//sampleEnd
}
```

You can get the full code from here.

The concurrent nature of flatMapMerge is obvious:

```
1: First at 136 ms from start
2: First at 231 ms from start
3: First at 333 ms from start
1: Second at 639 ms from start
2: Second at 732 ms from start
3: Second at 833 ms from start
```

Note that the flatMapMerge calls its block of code ({ requestFlow(it) } in this example) sequentially, but collects the resulting flows concurrently, it is the equivalent of performing a sequential map { requestFlow(it) } first and then calling flattenMerge on the result.

flatMapLatest

In a similar way to the collectLatest operator, that was described in the section "Processing the latest value", there is the corresponding "Latest" flattening mode where the collection of the previous flow is cancelled as soon as new flow is emitted. It is implemented by the flatMapLatest operator.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // wait 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // remember the start time
    (1..3).asFlow().onEach { delay(100) } // a number every 100 ms
        .flatMapLatest { requestFlow(it) }
        .collect { value -> // collect and print
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
//sampleEnd
}
```

You can get the full code from here.

The output here in this example is a good demonstration of how flatMapLatest works:

```
1: First at 142 ms from start
2: First at 322 ms from start
3: First at 425 ms from start
3: Second at 931 ms from start
```

Note that flatMapLatest cancels all the code in its block ({ requestFlow(it) } in this example) when a new value is received. It makes no difference in this particular example, because the call to requestFlow itself is fast, not-suspending, and cannot be cancelled. However, a differnce in output would be visible if we were to use suspending functions like delay in requestFlow.

Flow exceptions

Flow collection can complete with an exception when an emitter or code inside the operators throw an exception. There are several ways to handle these exceptions.

Collector try and catch

A collector can use Kotlin's try/catch block to handle exceptions:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    try {
```

```
        simple().collect { value ->
            println(value)
            check(value <= 1) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
//sampleEnd
```

You can get the full code from here.

This code successfully catches an exception in collect terminal operator and, as we see, no more values are emitted after that:

```
Emitting 1
1
Emitting 2
2
Caught java.lang.IllegalStateException: Collected 2
```

Everything is caught

The previous example actually catches any exception happening in the emitter or in any intermediate or terminal operators. For example, let's change the code so that emitted values are mapped to strings, but the corresponding code produces an exception:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // emit next value
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
//sampleEnd
```

You can get the full code from here.

This exception is still caught and collection is stopped:

```
Emitting 1
string 1
Emitting 2
Caught java.lang.IllegalStateException: Crashed on 2
```

Exception transparency

But how can code of the emitter encapsulate its exception handling behavior?

Flows must be transparent to exceptions and it is a violation of the exception transparency to emit values in the flow { ... } builder from inside of a try/catch block. This guarantees that a collector throwing an exception can always catch it using try/catch as in the previous example.

The emitter can use a catch operator that preserves this exception transparency and allows encapsulation of its exception handling. The body of the catch operator can analyze an exception and react to it in different ways depending on which exception was caught:

- Exceptions can be rethrown using throw.
- Exceptions can be turned into emission of values using emit from the body of catch.
- Exceptions can be ignored, logged, or processed by some other code.

For example, let us emit the text on catching an exception:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // emit next value
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .catch { e -> emit("Caught $e") } // emit on exception
        .collect { value -> println(value) }
//sampleEnd
}
```

> You can get the full code from here.

The output of the example is the same, even though we do not have try/catch around the code anymore.

Transparent catch

The catch intermediate operator, honoring exception transparency, catches only upstream exceptions (that is an exception from all the operators above catch, but not below it). If the block in collect { ... } (placed below catch) throws an exception then it escapes:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple()
        .catch { e -> println("Caught $e") } // does not catch downstream exceptions
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
//sampleEnd
```

> You can get the full code from here.

A "Caught ..." message is not printed despite there being a catch operator:

```
Emitting 1
1
Emitting 2
Exception in thread "main" java.lang.IllegalStateException: Collected 2
 at ...
```

Catching declaratively

We can combine the declarative nature of the catch operator with a desire to handle all the exceptions, by moving the body of the collect operator into onEach and putting it before the catch operator. Collection of this flow must be triggered by a call to collect() without parameters:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .onEach { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
        .catch { e -> println("Caught $e") }
        .collect()
//sampleEnd
}
```

You can get the full code from here.

Now we can see that a "Caught ..." message is printed and so we can catch all the exceptions without explicitly using a try/catch block:

```
Emitting 1
1
Emitting 2
Caught java.lang.IllegalStateException: Collected 2
```

Flow completion

When flow collection completes (normally or exceptionally) it may need to execute an action. As you may have already noticed, it can be done in two ways: imperative or declarative.

Imperative finally block

In addition to try/catch, a collector can also use a finally block to execute an action upon collect completion.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } finally {
        println("Done")
    }
}
//sampleEnd
```

You can get the full code from here.

This code prints three numbers produced by the simple flow followed by a "Done" string:

```
1
2
3
Done
```

Declarative handling

For the declarative approach, flow has onCompletion intermediate operator that is invoked when the flow has completely collected.

The previous example can be rewritten using an onCompletion operator and produces the same output:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .onCompletion { println("Done") }
        .collect { value -> println(value) }
//sampleEnd
}
```

You can get the full code from here.

The key advantage of onCompletion is a nullable Throwable parameter of the lambda that can be used to determine whether the flow collection was completed normally or exceptionally. In the following example the simple flow throws an exception after emitting the number 1:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    emit(1)
    throw RuntimeException()
}

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> if (cause != null) println("Flow completed exceptionally") }
        .catch { cause -> println("Caught exception") }
        .collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from here.

As you may expect, it prints:

```
1
Flow completed exceptionally
Caught exception
```

The onCompletion operator, unlike catch, does not handle the exception. As we can see from the above example code, the exception still flows downstream. It will be delivered to further onCompletion operators and can be handled with a catch operator.

Successful completion

Another difference with catch operator is that onCompletion sees all exceptions and receives a null exception only on successful completion of the upstream flow (without cancellation or failure).

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> println("Flow completed with $cause") }
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
//sampleEnd
```

> You can get the full code from here.

We can see the completion cause is not null, because the flow was aborted due to downstream exception:

```
1
Flow completed with java.lang.IllegalStateException: Collected 2
Exception in thread "main" java.lang.IllegalStateException: Collected 2
```

Imperative versus declarative

Now we know how to collect flow, and handle its completion and exceptions in both imperative and declarative ways. The natural question here is, which approach is preferred and why? As a library, we do not advocate for any particular approach and believe that both options are valid and should be selected according to your own preferences and code style.

Launching flow

It is easy to use flows to represent asynchronous events that are coming from some source. In this case, we need an analogue of the addEventListener function that registers a piece of code with a reaction for incoming events and continues further work. The onEach operator can serve this role. However, onEach is an intermediate operator. We also need a terminal operator to collect the flow. Otherwise, just calling onEach has no effect.

If we use the collect terminal operator after onEach, then the code after it will wait until the flow is collected:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
// Imitate a flow of events
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }

fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .collect() // <--- Collecting the flow waits
    println("Done")
}
//sampleEnd
```

> You can get the full code from here.

As you can see, it prints:

```
Event: 1
Event: 2
Event: 3
Done
```

The launchIn terminal operator comes in handy here. By replacing collect with launchIn we can launch a collection of the flow in a separate coroutine, so that execution of further code immediately continues:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// Imitate a flow of events
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }

//sampleStart
fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .launchIn(this) // <--- Launching the flow in a separate coroutine
    println("Done")
}
//sampleEnd
```

> You can get the full code from here.

It prints:

```
Done
Event: 1
Event: 2
Event: 3
```

The required parameter to launchIn must specify a CoroutineScope in which the coroutine to collect the flow is launched. In the above example this scope comes from the runBlocking coroutine builder, so while the flow is running, this runBlocking scope waits for completion of its child coroutine and keeps the main function from returning and terminating this example.

In actual applications a scope will come from an entity with a limited lifetime. As soon as the lifetime of this entity is terminated the corresponding scope is cancelled, cancelling the collection of the corresponding flow. This way the pair of onEach { ... }.launchIn(scope) works like the addEventListener. However, there is no need for the corresponding removeEventListener function, as cancellation and structured concurrency serve this purpose.

Note that launchIn also returns a Job, which can be used to cancel the corresponding flow collection coroutine only without cancelling the whole scope or to join it.

Flow cancellation checks

For convenience, the flow builder performs additional ensureActive checks for cancellation on each emitted value. It means that a busy loop emitting from a flow { ... } is cancellable:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun foo(): Flow<Int> = flow {
    for (i in 1..5) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    foo().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

> You can get the full code from here.

We get only numbers up to 3 and a CancellationException after trying to emit number 4:

```
Emitting 1
1
Emitting 2
2
Emitting 3
3
Emitting 4
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was cancelled;
job="coroutine#1":BlockingCoroutine{Cancelled}@6d7b4f4c
```

However, most other flow operators do not do additional cancellation checks on their own for performance reasons. For example, if you use IntRange.asFlow extension to write the same busy loop and don't suspend anywhere, then there are no checks for cancellation:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun main() = runBlocking<Unit> {
    (1..5).asFlow().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

> You can get the full code from here.

All numbers from 1 to 5 are collected and cancellation gets detected only before return from runBlocking:

```
1
2
3
4
5
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was cancelled;
job="coroutine#1":BlockingCoroutine{Cancelled}@3327bd23
```

Making busy flow cancellable

In the case where you have a busy loop with coroutines you must explicitly check for cancellation. You can add .onEach { currentCoroutineContext().ensureActive() }, but there is a ready-to-use cancellable operator provided to do that:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun main() = runBlocking<Unit> {
    (1..5).asFlow().cancellable().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

> You can get the full code from here.

With the cancellable operator only the numbers from 1 to 3 are collected:

```
1
```

```
2
3
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was cancelled;
job="coroutine#1":BlockingCoroutine{Cancelled}@5ec0a365
```

Flow and Reactive Streams

For those who are familiar with Reactive Streams or reactive frameworks such as RxJava and project Reactor, design of the Flow may look very familiar.

Indeed, its design was inspired by Reactive Streams and its various implementations. But Flow main goal is to have as simple design as possible, be Kotlin and suspension friendly and respect structured concurrency. Achieving this goal would be impossible without reactive pioneers and their tremendous work. You can read the complete story in Reactive Streams and Kotlin Flows article.

While being different, conceptually, Flow is a reactive stream and it is possible to convert it to the reactive (spec and TCK compliant) Publisher and vice versa. Such converters are provided by kotlinx.coroutines out-of-the-box and can be found in corresponding reactive modules (kotlinx-coroutines-reactive for Reactive Streams, kotlinx-coroutines-reactor for Project Reactor and kotlinx-coroutines-rx2/kotlinx-coroutines-rx3 for RxJava2/RxJava3). Integration modules include conversions from and to Flow, integration with Reactor's Context and suspension-friendly ways to work with various reactive entities.

Channels

Deferred values provide a convenient way to transfer a single value between coroutines. Channels provide a way to transfer a stream of values.

Channel basics

A Channel is conceptually very similar to BlockingQueue. One key difference is that instead of a blocking put operation it has a suspending send, and instead of a blocking take operation it has a suspending receive.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<Int>()
    launch {
        // this might be heavy CPU-consuming computation or async logic, we'll just send five squares
        for (x in 1..5) channel.send(x * x)
    }
    // here we print five received integers:
    repeat(5) { println(channel.receive()) }
    println("Done!")
//sampleEnd
}
```

You can get the full code here.

The output of this code is:

```
1
4
9
16
25
Done!
```

Closing and iteration over channels

Unlike a queue, a channel can be closed to indicate that no more elements are coming. On the receiver side it is convenient to use a regular for loop to receive elements from the channel.

Conceptually, a close is like sending a special close token to the channel. The iteration stops as soon as this close token is received, so there is a guarantee that all

previously sent elements before the close are received:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<Int>()
    launch {
        for (x in 1..5) channel.send(x * x)
        channel.close() // we're done sending
    }
    // here we print received values using `for` loop (until the channel is closed)
    for (y in channel) println(y)
    println("Done!")
//sampleEnd
}
```

You can get the full code here.

Building channel producers

The pattern where a coroutine is producing a sequence of elements is quite common. This is a part of producer-consumer pattern that is often found in concurrent code. You could abstract such a producer into a function that takes channel as its parameter, but this goes contrary to common sense that results must be returned from functions.

There is a convenient coroutine builder named produce that makes it easy to do it right on producer side, and an extension function consumeEach, that replaces a for loop on the consumer side:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun CoroutineScope.produceSquares(): ReceiveChannel<Int> = produce {
    for (x in 1..5) send(x * x)
}

fun main() = runBlocking {
//sampleStart
    val squares = produceSquares()
    squares.consumeEach { println(it) }
    println("Done!")
//sampleEnd
}
```

You can get the full code here.

Pipelines

A pipeline is a pattern where one coroutine is producing, possibly infinite, stream of values:

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // infinite stream of integers starting from 1
}
```

And another coroutine or coroutines are consuming that stream, doing some processing, and producing some other results. In the example below, the numbers are just squared:

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

The main code starts and connects the whole pipeline:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val numbers = produceNumbers() // produces integers from 1 and on
    val squares = square(numbers) // squares integers
    repeat(5) {
        println(squares.receive()) // print first five
    }
    println("Done!") // we are done
    coroutineContext.cancelChildren() // cancel children coroutines
//sampleEnd
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // infinite stream of integers starting from 1
}

fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

You can get the full code here.

All functions that create coroutines are defined as extensions on CoroutineScope, so that we can rely on structured concurrency to make sure that we don't have lingering global coroutines in our application.

Prime numbers with pipeline

Let's take pipelines to the extreme with an example that generates prime numbers using a pipeline of coroutines. We start with an infinite sequence of numbers.

```kotlin
fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // infinite stream of integers from start
}
```

The following pipeline stage filters an incoming stream of numbers, removing all the numbers that are divisible by the given prime number:

```kotlin
fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

Now we build our pipeline by starting a stream of numbers from 2, taking a prime number from the current channel, and launching new pipeline stage for each prime number found:

```
numbersFrom(2) -> filter(2) -> filter(3) -> filter(5) -> filter(7) ...
```

The following example prints the first ten prime numbers, running the whole pipeline in the context of the main thread. Since all the coroutines are launched in the scope of the main runBlocking coroutine we don't have to keep an explicit list of all the coroutines we have started. We use cancelChildren extension function to cancel all the children coroutines after we have printed the first ten prime numbers.

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    var cur = numbersFrom(2)
    repeat(10) {
        val prime = cur.receive()
        println(prime)
        cur = filter(cur, prime)
    }
    coroutineContext.cancelChildren() // cancel all children to let main finish
```

```
//sampleEnd
}

fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // infinite stream of integers from start
}

fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

You can get the full code here.

The output of this code is:

```
2
3
5
7
11
13
17
19
23
29
```

Note that you can build the same pipeline using iterator coroutine builder from the standard library. Replace produce with iterator, send with yield, receive with next, ReceiveChannel with Iterator, and get rid of the coroutine scope. You will not need runBlocking either. However, the benefit of a pipeline that uses channels as shown above is that it can actually use multiple CPU cores if you run it in Dispatchers.Default context.

Anyway, this is an extremely impractical way to find prime numbers. In practice, pipelines do involve some other suspending invocations (like asynchronous calls to remote services) and these pipelines cannot be built using sequence/iterator, because they do not allow arbitrary suspension, unlike produce, which is fully asynchronous.

Fan-out

Multiple coroutines may receive from the same channel, distributing work between themselves. Let us start with a producer coroutine that is periodically producing integers (ten numbers per second):

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // start from 1
    while (true) {
        send(x++) // produce next
        delay(100) // wait 0.1s
    }
}
```

Then we can have several processor coroutines. In this example, they just print their id and received number:

```
fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

Now let us launch five processors and let them work for almost a second. See what happens:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
//sampleStart
    val producer = produceNumbers()
    repeat(5) { launchProcessor(it, producer) }
    delay(950)
    producer.cancel() // cancel producer coroutine and thus kill them all
```

```
//sampleEnd
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // start from 1
    while (true) {
        send(x++) // produce next
        delay(100) // wait 0.1s
    }
}

fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

You can get the full code here.

The output will be similar to the the following one, albeit the processor ids that receive each specific integer may be different:

```
Processor #2 received 1
Processor #4 received 2
Processor #0 received 3
Processor #1 received 4
Processor #3 received 5
Processor #2 received 6
Processor #4 received 7
Processor #0 received 8
Processor #1 received 9
Processor #3 received 10
```

Note that cancelling a producer coroutine closes its channel, thus eventually terminating iteration over the channel that processor coroutines are doing.

Also, pay attention to how we explicitly iterate over channel with for loop to perform fan-out in launchProcessor code. Unlike consumeEach, this for loop pattern is perfectly safe to use from multiple coroutines. If one of the processor coroutines fails, then others would still be processing the channel, while a processor that is written via consumeEach always consumes (cancels) the underlying channel on its normal or abnormal completion.

Fan-in

Multiple coroutines may send to the same channel. For example, let us have a channel of strings, and a suspending function that repeatedly sends a specified string to this channel with a specified delay:

```
suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

Now, let us see what happens if we launch a couple of coroutines sending strings (in this example we launch them in the context of the main thread as main coroutine's children):

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<String>()
    launch { sendString(channel, "foo", 200L) }
    launch { sendString(channel, "BAR!", 500L) }
    repeat(6) { // receive first six
        println(channel.receive())
    }
    coroutineContext.cancelChildren() // cancel all children to let main finish
//sampleEnd
}

suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
```

```
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

You can get the full code here.

The output is:

```
foo
foo
BAR!
foo
foo
BAR!
```

Buffered channels

The channels shown so far had no buffer. Unbuffered channels transfer elements when sender and receiver meet each other (aka rendezvous). If send is invoked first, then it is suspended until receive is invoked, if receive is invoked first, it is suspended until send is invoked.

Both Channel() factory function and produce builder take an optional capacity parameter to specify buffer size. Buffer allows senders to send multiple elements before suspending, similar to the BlockingQueue with a specified capacity, which blocks when buffer is full.

Take a look at the behavior of the following code:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
//sampleStart
    val channel = Channel<Int>(4) // create buffered channel
    val sender = launch { // launch sender coroutine
        repeat(10) {
            println("Sending $it") // print before sending each element
            channel.send(it) // will suspend when buffer is full
        }
    }
    // don't receive anything... just wait....
    delay(1000)
    sender.cancel() // cancel sender coroutine
//sampleEnd
}
```

You can get the full code here.

It prints "sending" five times using a buffered channel with capacity of four:

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

The first four elements are added to the buffer and the sender suspends when trying to send the fifth one.

Channels are fair

Send and receive operations to channels are fair with respect to the order of their invocation from multiple coroutines. They are served in first-in first-out order, e.g. the first coroutine to invoke receive gets the element. In the following example two coroutines "ping" and "pong" are receiving the "ball" object from the shared "table" channel.

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

//sampleStart
data class Ball(var hits: Int)

fun main() = runBlocking {
    val table = Channel<Ball>() // a shared table
    launch { player("ping", table) }
    launch { player("pong", table) }
    table.send(Ball(0)) // serve the ball
    delay(1000) // delay 1 second
    coroutineContext.cancelChildren() // game over, cancel them
}

suspend fun player(name: String, table: Channel<Ball>) {
    for (ball in table) { // receive the ball in a loop
        ball.hits++
        println("$name $ball")
        delay(300) // wait a bit
        table.send(ball) // send the ball back
    }
}
//sampleEnd
```

You can get the full code here.

The "ping" coroutine is started first, so it is the first one to receive the ball. Even though "ping" coroutine immediately starts receiving the ball again after sending it back to the table, the ball gets received by the "pong" coroutine, because it was already waiting for it:

```
ping Ball(hits=1)
pong Ball(hits=2)
ping Ball(hits=3)
pong Ball(hits=4)
```

Note that sometimes channels may produce executions that look unfair due to the nature of the executor that is being used. See this issue for details.

Ticker channels

Ticker channel is a special rendezvous channel that produces Unit every time given delay passes since last consumption from this channel. Though it may seem to be useless standalone, it is a useful building block to create complex time-based produce pipelines and operators that do windowing and other time-dependent processing. Ticker channel can be used in select to perform "on tick" action.

To create such channel use a factory method ticker. To indicate that no further elements are needed use ReceiveChannel.cancel method on it.

Now let's see how it works in practice:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

//sampleStart
fun main() = runBlocking<Unit> {
    val tickerChannel = ticker(delayMillis = 100, initialDelayMillis = 0) // create ticker channel
    var nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Initial element is available immediately: $nextElement") // no initial delay

    nextElement = withTimeoutOrNull(50) { tickerChannel.receive() } // all subsequent elements have 100ms delay
    println("Next element is not ready in 50 ms: $nextElement")

    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 100 ms: $nextElement")

    // Emulate large consumption delays
    println("Consumer pauses for 150ms")
    delay(150)
    // Next element is available immediately
    nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Next element is available immediately after large consumer delay: $nextElement")
    // Note that the pause between `receive` calls is taken into account and next element arrives faster
    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
```

```
    println("Next element is ready in 50ms after consumer pause in 150ms: $nextElement")

    tickerChannel.cancel() // indicate that no more elements are needed
}
//sampleEnd
```

> You can get the full code here.

It prints following lines:

```
Initial element is available immediately: kotlin.Unit
Next element is not ready in 50 ms: null
Next element is ready in 100 ms: kotlin.Unit
Consumer pauses for 150ms
Next element is available immediately after large consumer delay: kotlin.Unit
Next element is ready in 50ms after consumer pause in 150ms: kotlin.Unit
```

Note that ticker is aware of possible consumer pauses and, by default, adjusts next produced element delay if a pause occurs, trying to maintain a fixed rate of produced elements.

Optionally, a mode parameter equal to TickerMode.FIXED_DELAY can be specified to maintain a fixed delay between elements.

Coroutine exceptions handling

This section covers exception handling and cancellation on exceptions. We already know that a cancelled coroutine throws CancellationException in suspension points and that it is ignored by the coroutines' machinery. Here we look at what happens if an exception is thrown during cancellation or multiple children of the same coroutine throw an exception.

Exception propagation

Coroutine builders come in two flavors: propagating exceptions automatically (launch and actor) or exposing them to users (async and produce). When these builders are used to create a root coroutine, that is not a child of another coroutine, the former builders treat exceptions as uncaught exceptions, similar to Java's Thread.uncaughtExceptionHandler, while the latter are relying on the user to consume the final exception, for example via await or receive (produce and receive are covered in Channels section).

It can be demonstrated by a simple example that creates root coroutines using the GlobalScope:

> GlobalScope is a delicate API that can backfire in non-trivial ways. Creating a root coroutine for the whole application is one of the rare legitimate uses for GlobalScope, so you must explicitly opt-in into using GlobalScope with @OptIn(DelicateCoroutinesApi::class).

```
import kotlinx.coroutines.*

//sampleStart
@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val job = GlobalScope.launch { // root coroutine with launch
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException() // Will be printed to the console by Thread.defaultUncaughtExceptionHandler
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async { // root coroutine with async
        println("Throwing exception from async")
        throw ArithmeticException() // Nothing is printed, relying on user to call await
    }
    try {
        deferred.await()
        println("Unreached")
    } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
    }
}
//sampleEnd
```

> You can get the full code here.

The output of this code is (with debug):

```
Throwing exception from launch
Exception in thread "DefaultDispatcher-worker-2 @coroutine#2" java.lang.IndexOutOfBoundsException
Joined failed job
Throwing exception from async
Caught ArithmeticException
```

CoroutineExceptionHandler

It is possible to customize the default behavior of printing uncaught exceptions to the console. CoroutineExceptionHandler context element on a root coroutine can be used as a generic catch block for this root coroutine and all its children where custom exception handling may take place. It is similar to Thread.uncaughtExceptionHandler. You cannot recover from the exception in the CoroutineExceptionHandler. The coroutine had already completed with the corresponding exception when the handler is called. Normally, the handler is used to log the exception, show some kind of error message, terminate, and/or restart the application.

CoroutineExceptionHandler is invoked only on uncaught exceptions — exceptions that were not handled in any other way. In particular, all children coroutines (coroutines created in the context of another Job) delegate handling of their exceptions to their parent coroutine, which also delegates to the parent, and so on until the root, so the CoroutineExceptionHandler installed in their context is never used. In addition to that, async builder always catches all exceptions and represents them in the resulting Deferred object, so its CoroutineExceptionHandler has no effect either.

> Coroutines running in supervision scope do not propagate exceptions to their parent and are excluded from this rule. A further Supervision section of this document gives more details.

```kotlin
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) { // root coroutine, running in GlobalScope
        throw AssertionError()
    }
    val deferred = GlobalScope.async(handler) { // also root, but async instead of launch
        throw ArithmeticException() // Nothing will be printed, relying on user to call deferred.await()
    }
    joinAll(job, deferred)
//sampleEnd
}
```

> You can get the full code here.

The output of this code is:

```
CoroutineExceptionHandler got java.lang.AssertionError
```

Cancellation and exceptions

Cancellation is closely related to exceptions. Coroutines internally use CancellationException for cancellation, these exceptions are ignored by all handlers, so they should be used only as the source of additional debug information, which can be obtained by catch block. When a coroutine is cancelled using Job.cancel, it terminates, but it does not cancel its parent.

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        val child = launch {
            try {
                delay(Long.MAX_VALUE)
            } finally {
                println("Child is cancelled")
            }
        }
        yield()
        println("Cancelling child")
        child.cancel()
        child.join()
        yield()
        println("Parent is not cancelled")
    }
    job.join()
//sampleEnd
}
```

> You can get the full code here.

The output of this code is:

```
Cancelling child
Child is cancelled
Parent is not cancelled
```

If a coroutine encounters an exception other than CancellationException, it cancels its parent with that exception. This behaviour cannot be overridden and is used to provide stable coroutines hierarchies for structured concurrency. CoroutineExceptionHandler implementation is not used for child coroutines.

> In these examples, CoroutineExceptionHandler is always installed to a coroutine that is created in GlobalScope. It does not make sense to install an exception handler to a coroutine that is launched in the scope of the main runBlocking, since the main coroutine is going to be always cancelled when its child completes with exception despite the installed handler.

The original exception is handled by the parent only when all its children terminate, which is demonstrated by the following example.

```kotlin
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        launch { // the first child
            try {
                delay(Long.MAX_VALUE)
            } finally {
                withContext(NonCancellable) {
                    println("Children are cancelled, but exception is not handled until all children terminate")
                    delay(100)
                    println("The first child finished its non cancellable block")
                }
            }
        }
        launch { // the second child
            delay(10)
            println("Second child throws an exception")
            throw ArithmeticException()
        }
    }
    job.join()
//sampleEnd
}
```

You can get the full code here.

The output of this code is:

```
Second child throws an exception
Children are cancelled, but exception is not handled until all children terminate
The first child finished its non cancellable block
CoroutineExceptionHandler got java.lang.ArithmeticException
```

Exceptions aggregation

When multiple children of a coroutine fail with an exception, the general rule is "the first exception wins", so the first exception gets handled. All additional exceptions that happen after the first one are attached to the first exception as suppressed ones.

```
import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception with suppressed ${exception.suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
                delay(Long.MAX_VALUE) // it gets cancelled when another sibling fails with IOException
            } finally {
                throw ArithmeticException() // the second exception
            }
        }
        launch {
            delay(100)
            throw IOException() // the first exception
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}
```

You can get the full code here.

Note: This above code will work properly only on JDK7+ that supports suppressed exceptions

The output of this code is:

```
CoroutineExceptionHandler got java.io.IOException with suppressed [java.lang.ArithmeticException]
```

Note that this mechanism currently only works on Java version 1.7+. The JS and Native restrictions are temporary and will be lifted in the future.

Cancellation exceptions are transparent and are unwrapped by default:

```
import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
```

```
        val inner = launch { // all this stack of coroutines will get cancelled
            launch {
                launch {
                    throw IOException() // the original exception
                }
            }
        }
        try {
            inner.join()
        } catch (e: CancellationException) {
            println("Rethrowing CancellationException with original cause")
            throw e // cancellation exception is rethrown, yet the original IOException gets to the handler
        }
    }
    job.join()
//sampleEnd
}
```

You can get the full code here.

The output of this code is:

```
Rethrowing CancellationException with original cause
CoroutineExceptionHandler got java.io.IOException
```

Supervision

As we have studied before, cancellation is a bidirectional relationship propagating through the whole hierarchy of coroutines. Let us take a look at the case when unidirectional cancellation is required.

A good example of such a requirement is a UI component with the job defined in its scope. If any of the UI's child tasks have failed, it is not always necessary to cancel (effectively kill) the whole UI component, but if the UI component is destroyed (and its job is cancelled), then it is necessary to cancel all child jobs as their results are no longer needed.

Another example is a server process that spawns multiple child jobs and needs to supervise their execution, tracking their failures and only restarting the failed ones.

Supervision job

The SupervisorJob can be used for these purposes. It is similar to a regular Job with the only exception that cancellation is propagated only downwards. This can easily be demonstrated using the following example:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        // launch the first child -- its exception is ignored for this example (don't do this in practice!)
        val firstChild = launch(CoroutineExceptionHandler { _, _ ->  }) {
            println("The first child is failing")
            throw AssertionError("The first child is cancelled")
        }
        // launch the second child
        val secondChild = launch {
            firstChild.join()
            // Cancellation of the first child is not propagated to the second child
            println("The first child is cancelled: ${firstChild.isCancelled}, but the second one is still active")
            try {
                delay(Long.MAX_VALUE)
            } finally {
                // But cancellation of the supervisor is propagated
                println("The second child is cancelled because the supervisor was cancelled")
            }
        }
        // wait until the first child fails & completes
        firstChild.join()
        println("Cancelling the supervisor")
        supervisor.cancel()
```

```
        secondChild.join()
    }
//sampleEnd
}
```

You can get the full code here.

The output of this code is:

```
The first child is failing
The first child is cancelled: true, but the second one is still active
Cancelling the supervisor
The second child is cancelled because the supervisor was cancelled
```

Supervision scope

Instead of coroutineScope, we can use supervisorScope for scoped concurrency. It propagates the cancellation in one direction only and cancels all its children only if it failed itself. It also waits for all children before completion just like coroutineScope does.

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("The child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("The child is cancelled")
                }
            }
            // Give our child a chance to execute and print using yield
            yield()
            println("Throwing an exception from the scope")
            throw AssertionError()
        }
    } catch(e: AssertionError) {
        println("Caught an assertion error")
    }
//sampleEnd
}
```

You can get the full code here.

The output of this code is:

```
The child is sleeping
Throwing an exception from the scope
The child is cancelled
Caught an assertion error
```

Exceptions in supervised coroutines

Another crucial difference between regular and supervisor jobs is exception handling. Every child should handle its exceptions by itself via the exception handling mechanism. This difference comes from the fact that child's failure does not propagate to the parent. It means that coroutines launched directly inside the supervisorScope do use the CoroutineExceptionHandler that is installed in their scope in the same way as root coroutines do (see the CoroutineExceptionHandler section for details).

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
```

```
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("The child throws an exception")
            throw AssertionError()
        }
        println("The scope is completing")
    }
    println("The scope is completed")
//sampleEnd
}
```

You can get the full code here.

The output of this code is:

```
The scope is completing
The child throws an exception
CoroutineExceptionHandler got java.lang.AssertionError
The scope is completed
```

Shared mutable state and concurrency

Coroutines can be executed parallelly using a multi-threaded dispatcher like the Dispatchers.Default. It presents all the usual parallelism problems. The main problem being synchronization of access to shared mutable state. Some solutions to this problem in the land of coroutines are similar to the solutions in the multi-threaded world, but others are unique.

The problem

Let us launch a hundred coroutines all doing the same action a thousand times. We'll also measure their completion time for further comparisons:

```
suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

We start with a very simple action that increments a shared mutable variable using multi-threaded Dispatchers.Default.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
```

```
}

//sampleStart
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

You can get the full code here.

What does it print at the end? It is highly unlikely to ever print "Counter = 100000", because a hundred coroutines increment the counter concurrently from multiple threads without any synchronization.

Volatiles are of no help

There is a common misconception that making a variable volatile solves concurrency problem. Let us try it:

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
@Volatile // in Kotlin `volatile` is an annotation
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

You can get the full code here.

This code works slower, but we still don't always get "Counter = 100000" at the end, because volatile variables guarantee linearizable (this is a technical term for "atomic") reads and writes to the corresponding variable, but do not provide atomicity of larger actions (increment in our case).

Thread-safe data structures

The general solution that works both for threads and for coroutines is to use a thread-safe (aka synchronized, linearizable, or atomic) data structure that provides all the necessary synchronization for the corresponding operations that needs to be performed on a shared state. In the case of a simple counter we can use AtomicInteger class which has atomic incrementAndGet operations:

```kotlin
import kotlinx.coroutines.*
import java.util.concurrent.atomic.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counter = AtomicInteger()

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter.incrementAndGet()
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

You can get the full code here.

This is the fastest solution for this particular problem. It works for plain counters, collections, queues and other standard data structures and basic operations on them. However, it does not easily scale to complex state or to complex operations that do not have ready-to-use thread-safe implementations.

Thread confinement fine-grained

Thread confinement is an approach to the problem of shared mutable state where all access to the particular shared state is confined to a single thread. It is typically used in UI applications, where all UI state is confined to the single event-dispatch/application thread. It is easy to apply with coroutines by using a single-threaded context.

```kotlin
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // confine each increment to a single-threaded context
            withContext(counterContext) {
                counter++
            }
```

```
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

You can get the full code here.

This code works very slowly, because it does fine-grained thread-confinement. Each individual increment switches from multi-threaded Dispatchers.Default context to the single-threaded context using withContext(counterContext) block.

Thread confinement coarse-grained

In practice, thread confinement is performed in large chunks, e.g. big pieces of state-updating business logic are confined to the single thread. The following example does it like that, running each coroutine in the single-threaded context to start with.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    // confine everything to a single-threaded context
    withContext(counterContext) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

You can get the full code here.

This now works much faster and produces correct result.

Mutual exclusion

Mutual exclusion solution to the problem is to protect all modifications of the shared state with a critical section that is never executed concurrently. In a blocking world you'd typically use synchronized or ReentrantLock for that. Coroutine's alternative is called Mutex. It has lock and unlock functions to delimit a critical section. The key difference is that Mutex.lock() is a suspending function. It does not block a thread.

There is also withLock extension function that conveniently represents mutex.lock(); try { ... } finally { mutex.unlock() } pattern:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.sync.*
import kotlin.system.*
```

```kotlin
suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val mutex = Mutex()
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // protect each increment with lock
            mutex.withLock {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

You can get the full code here.

The locking in this example is fine-grained, so it pays the price. However, it is a good choice for some situations where you absolutely must modify some shared state periodically, but there is no natural thread that this state is confined to.

Select expression (experimental)

Select expression makes it possible to await multiple suspending functions simultaneously and select the first one that becomes available.

Select expressions are an experimental feature of kotlinx.coroutines. Their API is expected to evolve in the upcoming updates of the kotlinx.coroutines library with potentially breaking changes.

Selecting from channels

Let us have two producers of strings: fizz and buzz. The fizz produces "Fizz" string every 500 ms:

```kotlin
fun CoroutineScope.fizz() = produce<String> {
    while (true) { // sends "Fizz" every 500 ms
        delay(500)
        send("Fizz")
    }
}
```

And the buzz produces "Buzz!" string every 1000 ms:

```kotlin
fun CoroutineScope.buzz() = produce<String> {
    while (true) { // sends "Buzz!" every 1000 ms
        delay(1000)
        send("Buzz!")
    }
}
```

Using receive suspending function we can receive either from one channel or the other. But select expression allows us to receive from both simultaneously using

its onReceive clauses:

```
suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>) {
    select<Unit> { // <Unit> means that this select expression does not produce any result
        fizz.onReceive { value ->  // this is the first select clause
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // this is the second select clause
            println("buzz -> '$value'")
        }
    }
}
```

Let us run it all seven times:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.fizz() = produce<String> {
    while (true) { // sends "Fizz" every 500 ms
        delay(500)
        send("Fizz")
    }
}

fun CoroutineScope.buzz() = produce<String> {
    while (true) { // sends "Buzz!" every 1000 ms
        delay(1000)
        send("Buzz!")
    }
}

suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>) {
    select<Unit> { // <Unit> means that this select expression does not produce any result
        fizz.onReceive { value ->  // this is the first select clause
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // this is the second select clause
            println("buzz -> '$value'")
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val fizz = fizz()
    val buzz = buzz()
    repeat(7) {
        selectFizzBuzz(fizz, buzz)
    }
    coroutineContext.cancelChildren() // cancel fizz & buzz coroutines
//sampleEnd
}
```

You can get the full code here.

The result of this code is:

```
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
```

Selecting on close

The onReceive clause in select fails when the channel is closed causing the corresponding select to throw an exception. We can use onReceiveCatching clause to

perform a specific action when the channel is closed. The following example also shows that select is an expression that returns the result of its selected clause:

```
suspend fun selectAorB(a: ReceiveChannel<String>, b: ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "a -> '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "b -> '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }
```

Let's use it with channel a that produces "Hello" string four times and channel b that produces "World" four times:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

suspend fun selectAorB(a: ReceiveChannel<String>, b: ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "a -> '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "b -> '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }

fun main() = runBlocking<Unit> {
//sampleStart
    val a = produce<String> {
        repeat(4) { send("Hello $it") }
    }
    val b = produce<String> {
        repeat(4) { send("World $it") }
    }
    repeat(8) { // print first eight results
        println(selectAorB(a, b))
    }
    coroutineContext.cancelChildren()
//sampleEnd
}
```

You can get the full code here.

The result of this code is quite interesting, so we'll analyze it in more detail:

```
a -> 'Hello 0'
a -> 'Hello 1'
b -> 'World 0'
a -> 'Hello 2'
a -> 'Hello 3'
b -> 'World 1'
Channel 'a' is closed
```

```
Channel 'a' is closed
```

There are a couple of observations to make out of it.

First of all, select is biased to the first clause. When several clauses are selectable at the same time, the first one among them gets selected. Here, both channels are constantly producing strings, so a channel, being the first clause in select, wins. However, because we are using unbuffered channel, the a gets suspended from time to time on its send invocation and gives a chance for b to send, too.

The second observation, is that onReceiveCatching gets immediately selected when the channel is already closed.

Selecting to send

Select expression has onSend clause that can be used for a great good in combination with a biased nature of selection.

Let us write an example of a producer of integers that sends its values to a side channel when the consumers on its primary channel cannot keep up with it:

```
fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // produce 10 numbers from 1 to 10
        delay(100) // every 100 ms
        select<Unit> {
            onSend(num) {} // Send to the primary channel
            side.onSend(num) {} // or to the side channel
        }
    }
}
```

Consumer is going to be quite slow, taking 250 ms to process each number:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // produce 10 numbers from 1 to 10
        delay(100) // every 100 ms
        select<Unit> {
            onSend(num) {} // Send to the primary channel
            side.onSend(num) {} // or to the side channel
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val side = Channel<Int>() // allocate side channel
    launch { // this is a very fast consumer for the side channel
        side.consumeEach { println("Side channel has $it") }
    }
    produceNumbers(side).consumeEach {
        println("Consuming $it")
        delay(250) // let us digest the consumed number properly, do not hurry
    }
    println("Done consuming")
    coroutineContext.cancelChildren()
//sampleEnd
}
```

You can get the full code here.

So let us see what happens:

```
Consuming 1
Side channel has 2
Side channel has 3
Consuming 4
Side channel has 5
Side channel has 6
Consuming 7
Side channel has 8
```

```
Side channel has 9
Consuming 10
Done consuming
```

Selecting deferred values

Deferred values can be selected using onAwait clause. Let us start with an async function that returns a deferred string value after a random delay:

```
fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}
```

Let us start a dozen of them with a random delay.

```
fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}
```

Now the main function awaits for the first of them to complete and counts the number of deferred values that are still active. Note that we've used here the fact that select expression is a Kotlin DSL, so we can provide clauses for it using an arbitrary code. In this case we iterate over a list of deferred values to provide onAwait clause for each deferred value.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.selects.*
import java.util.*

fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}

fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val list = asyncStringsList()
    val result = select<String> {
        list.withIndex().forEach { (index, deferred) ->
            deferred.onAwait { answer ->
                "Deferred $index produced answer '$answer'"
            }
        }
    }
    println(result)
    val countActive = list.count { it.isActive }
    println("$countActive coroutines are still active")
//sampleEnd
}
```

You can get the full code here.

The output is:

```
Deferred 4 produced answer 'Waited for 128 ms'
11 coroutines are still active
```

Switch over a channel of deferred values

Let us write a channel producer function that consumes a channel of deferred string values, waits for each received deferred value, but only until the next deferred value comes over or the channel is closed. This example puts together onReceiveCatching and onAwait clauses in the same select:

```kotlin
fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = produce<String> {
    var current = input.receive() // start with first received deferred value
    while (isActive) { // loop while not cancelled/closed
        val next = select<Deferred<String>?> { // return next deferred value from this select or null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // send value that current deferred has produced
                input.receiveCatching().getOrNull() // and use the next deferred from the input channel
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // out of loop
        } else {
            current = next
        }
    }
}
```

To test it, we'll use a simple async function that resolves to a specified string after a specified time:

```kotlin
fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}
```

The main function just launches a coroutine to print results of switchMapDeferreds and sends some test data to it:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = produce<String> {
    var current = input.receive() // start with first received deferred value
    while (isActive) { // loop while not cancelled/closed
        val next = select<Deferred<String>?> { // return next deferred value from this select or null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // send value that current deferred has produced
                input.receiveCatching().getOrNull() // and use the next deferred from the input channel
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // out of loop
        } else {
            current = next
        }
    }
}

fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}

fun main() = runBlocking<Unit> {
//sampleStart
    val chan = Channel<Deferred<String>>() // the channel for test
    launch { // launch printing coroutine
        for (s in switchMapDeferreds(chan))
            println(s) // print each received string
    }
    chan.send(asyncString("BEGIN", 100))
    delay(200) // enough time for "BEGIN" to be produced
    chan.send(asyncString("Slow", 500))
    delay(100) // not enough time to produce slow
    chan.send(asyncString("Replace", 100))
    delay(500) // give it time before the last one
    chan.send(asyncString("END", 500))
    delay(1000) // give it time to process
    chan.close() // close the channel ...
    delay(500) // and wait some time to let it finish
```

```
//sampleEnd
}
```

You can get the full code here.

The result of this code:

```
BEGIN
Replace
END
Channel was closed
```

Debug coroutines using IntelliJ IDEA – tutorial

This tutorial demonstrates how to create Kotlin coroutines and debug them using IntelliJ IDEA.

The tutorial assumes you have prior knowledge of the coroutines concept.

Create coroutines

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, create one.
2. To use the kotlinx.coroutines library in a Gradle project, add the following dependency to build.gradle(.kts):

   Kotlin

   ```
   dependencies {
       implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
   }
   ```

   Groovy

   ```
   dependencies {
       implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1'
   }
   ```

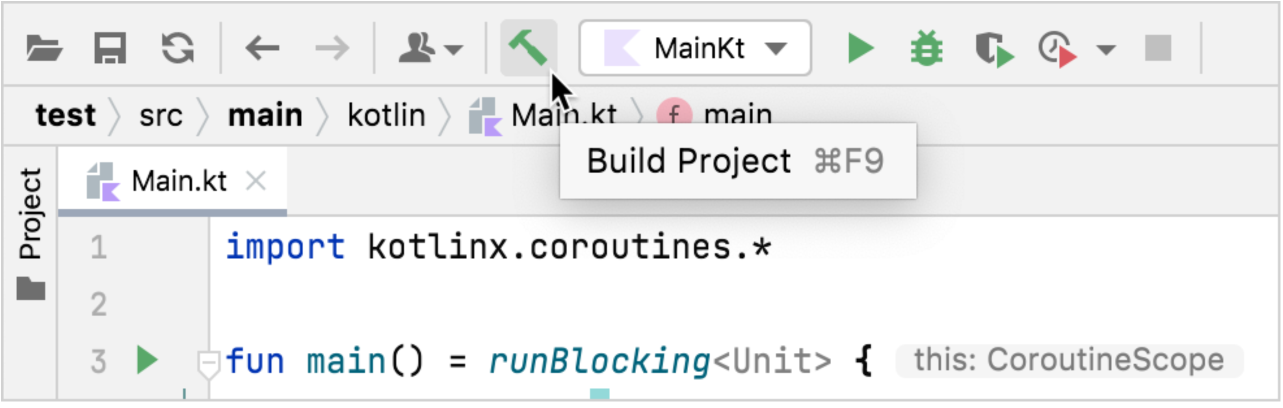
   For other build systems, see instructions in the kotlinx.coroutines README.
3. Open the Main.kt file in src/main/kotlin.

   The src directory contains Kotlin source files and resources. The Main.kt file contains sample code that will print Hello World!.
4. Change code in the main() function:
   - Use the runBlocking() block to wrap a coroutine.
   - Use the async() function to create coroutines that compute deferred values a and b.
   - Use the await() function to await the computation result.
   - Use the println() function to print computing status and the result of multiplication to the output.

   ```
   import kotlinx.coroutines.*

   fun main() = runBlocking<Unit> {
       val a = async {
           println("I'm computing part of the answer")
           6
       }
       val b = async {
           println("I'm computing another part of the answer")
   ```

```
        7
    }
    println("The answer is ${a.await() * b.await()}")
}
```

5. Build the code by clicking Build Project.



Build an application

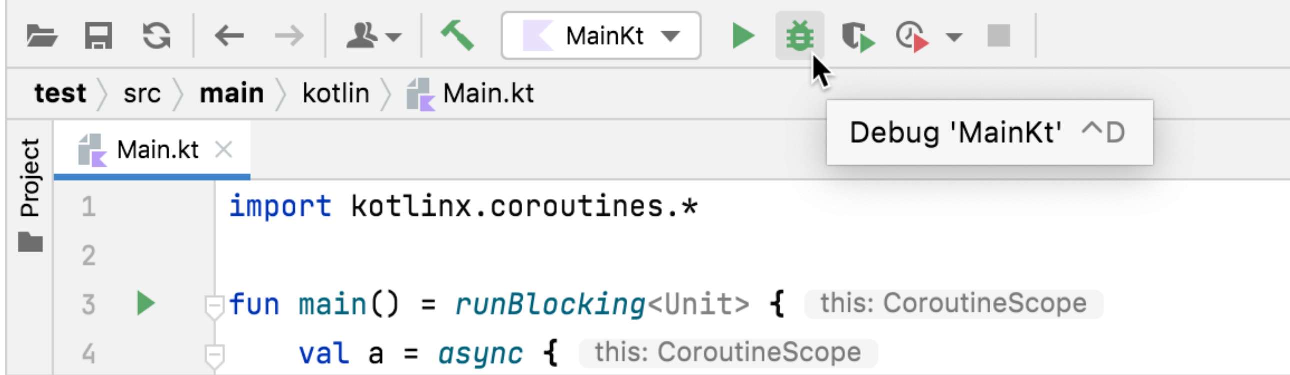
Debug coroutines

1. Set breakpoints at the lines with the println() function call:

```
Main.kt
1   import kotlinx.coroutines.*
2
3   fun main() = runBlocking<Unit> {  this: CoroutineScope
4       val a = async {  this: CoroutineScope
5           println("I'm computing part of the answer")
6           6  ^async
7       }
8       val b = async {  this: CoroutineScope
9           println("I'm computing another part of the answer")
10          7  ^async
11      }
12      println("The answer is ${a.await() * b.await()}")
13  }
14
```

Build a console application

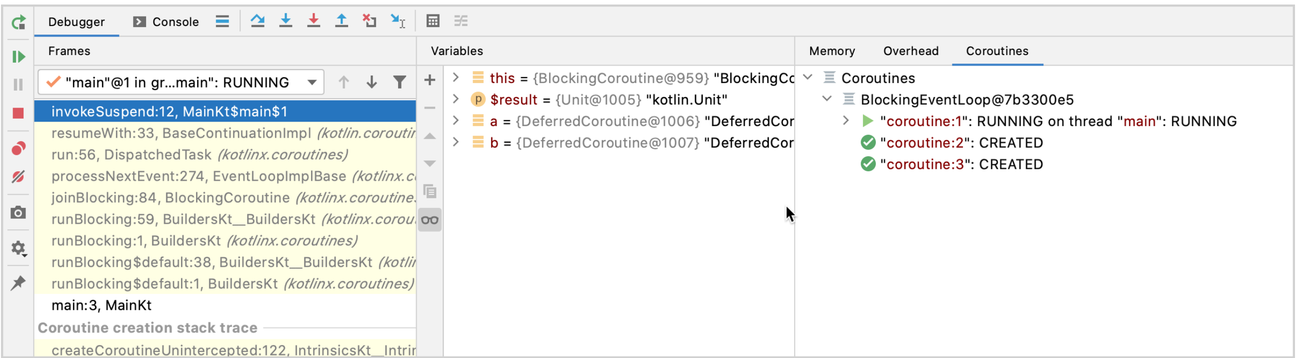
2. Run the code in debug mode by clicking Debug next to the run configuration at the top of the screen.



Build a console application

The Debug tool window appears:

- The Frames tab contains the call stack.
- The Variables tab contains variables in the current context.
- The Coroutines tab contains information on running or suspended coroutines. It shows that there are three coroutines. The first one has the RUNNING status, and the other two have the CREATED status.



Debug the coroutine

3. Resume the debugger session by clicking Resume Program in the Debug tool window:

Debug the coroutine

Now the Coroutines tab shows the following:

- The first coroutine has the SUSPENDED status – it is waiting for the values so it can multiply them.
- The second coroutine is calculating the a value – it has the RUNNING status.

* The third coroutine has the CREATED status and isn't calculating the value of b.

4. Resume the debugger session by clicking Resume Program in the Debug tool window:

Build a console application

Now the Coroutines tab shows the following:

* The first coroutine has the SUSPENDED status – it is waiting for the values so it can multiply them.
* The second coroutine has computed its value and disappeared.
* The third coroutine is calculating the value of b – it has the RUNNING status.

Using IntelliJ IDEA debugger, you can dig deeper into each coroutine to debug your code.

Optimized-out variables

If you use suspend functions, in the debugger, you might see the "was optimized out" text next to a variable's name:

Variable "a" was optimized out

This text means that the variable's lifetime was decreased, and the variable doesn't exist anymore. It is difficult to debug code with optimized variables because you

don't see their values. You can disable this behavior with the -Xdebug compiler option.

> Never use this flag in production: -Xdebug can cause memory leaks.

Debug Kotlin Flow using IntelliJ IDEA – tutorial

This tutorial demonstrates how to create Kotlin Flow and debug it using IntelliJ IDEA.

The tutorial assumes you have prior knowledge of the coroutines and Kotlin Flow concepts.

Create a Kotlin flow

Create a Kotlin flow with a slow emitter and a slow collector:

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, create one.

2. To use the kotlinx.coroutines library in a Gradle project, add the following dependency to build.gradle(.kts):

   Kotlin

   ```
   dependencies {
       implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
   }
   ```

   Groovy

   ```
   dependencies {
       implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1'
   }
   ```

   For other build systems, see instructions in the kotlinx.coroutines README.

3. Open the Main.kt file in src/main/kotlin.

   The src directory contains Kotlin source files and resources. The Main.kt file contains sample code that will print Hello World!.

4. Create the simple() function that returns a flow of three numbers:

   - Use the delay() function to imitate CPU-consuming blocking code. It suspends the coroutine for 100 ms without blocking the thread.

   - Produce the values in the for loop using the emit() function.

   ```
   import kotlinx.coroutines.*
   import kotlinx.coroutines.flow.*
   import kotlin.system.*

   fun simple(): Flow<Int> = flow {
       for (i in 1..3) {
           delay(100)
           emit(i)
       }
   }
   ```

5. Change the code in the main() function:

   - Use the runBlocking() block to wrap a coroutine.

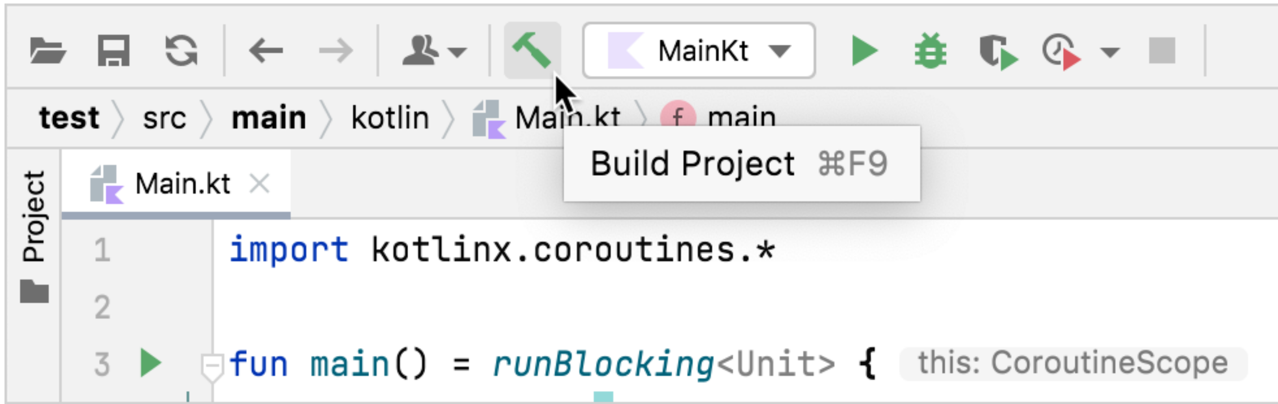
   - Collect the emitted values using the collect() function.

   - Use the delay() function to imitate CPU-consuming code. It suspends the coroutine for 300 ms without blocking the thread.

   - Print the collected value from the flow using the println() function.

```
fun main() = runBlocking {
    simple()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

6. Build the code by clicking Build Project.



Build an application

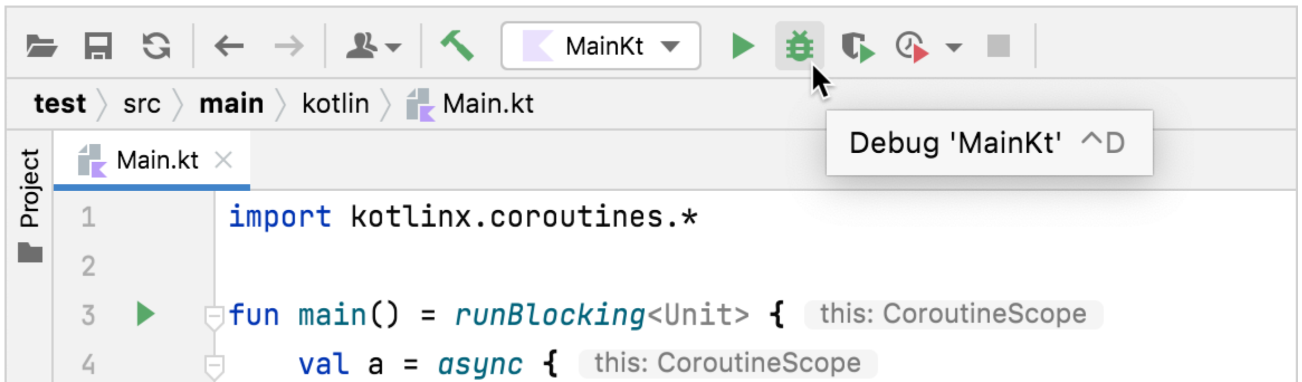
Debug the coroutine

1. Set a breakpoint at the line where the emit() function is called:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking {
    simple()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

Build a console application

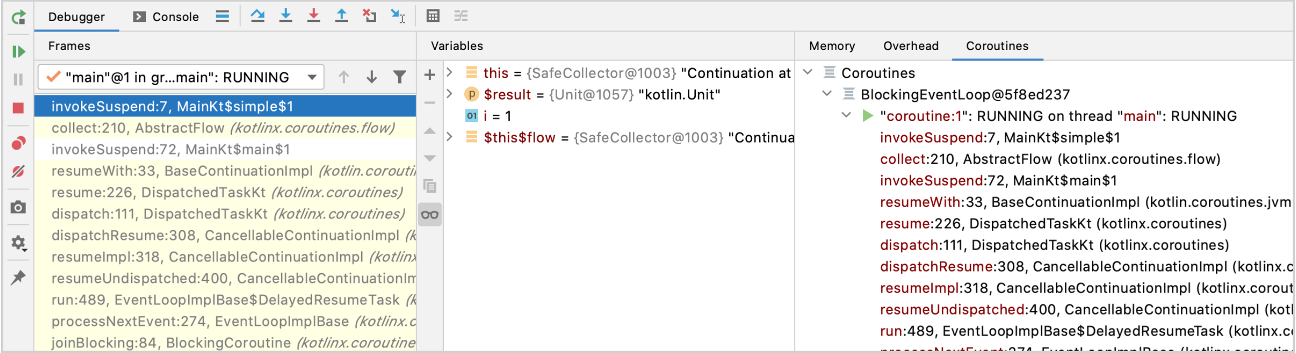
2. Run the code in debug mode by clicking Debug next to the run configuration at the top of the screen.



Build a console application

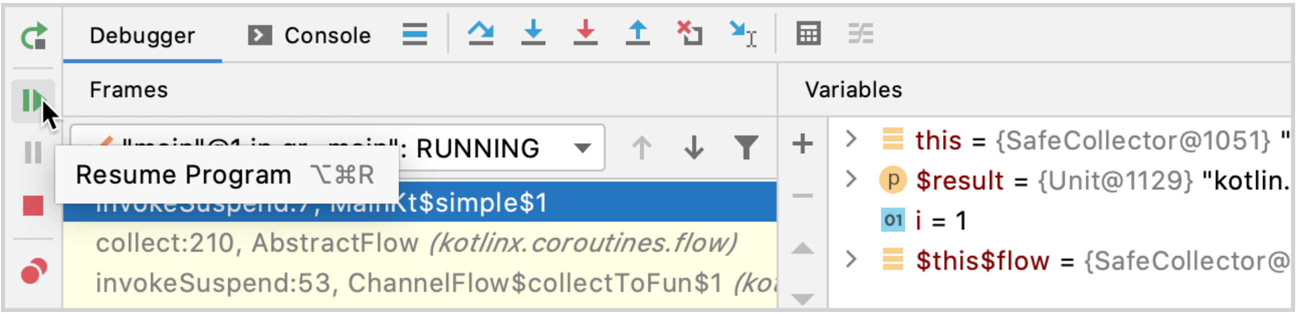
The Debug tool window appears:

- The Frames tab contains the call stack.
- The Variables tab contains variables in the current context. It tells us that the flow is emitting the first value.
- The Coroutines tab contains information on running or suspended coroutines.



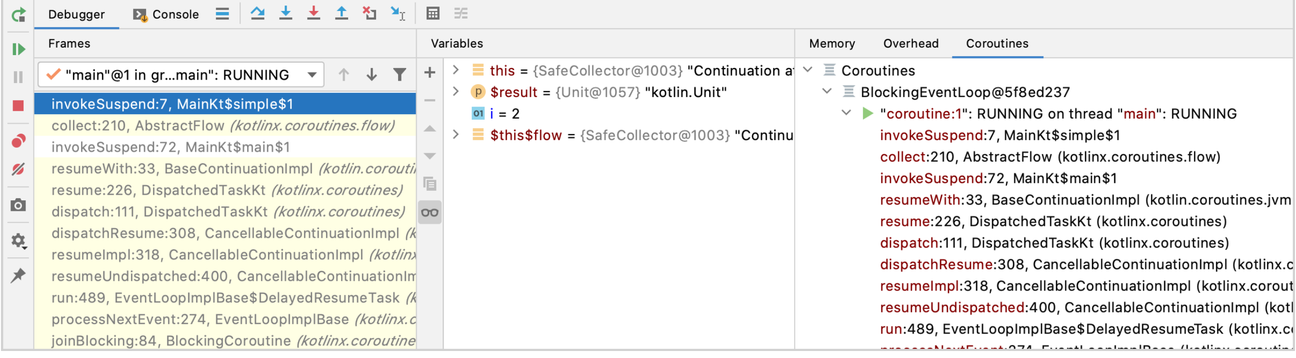
Debug the coroutine

3. Resume the debugger session by clicking Resume Program in the Debug tool window. The program stops at the same breakpoint.



Debug the coroutine

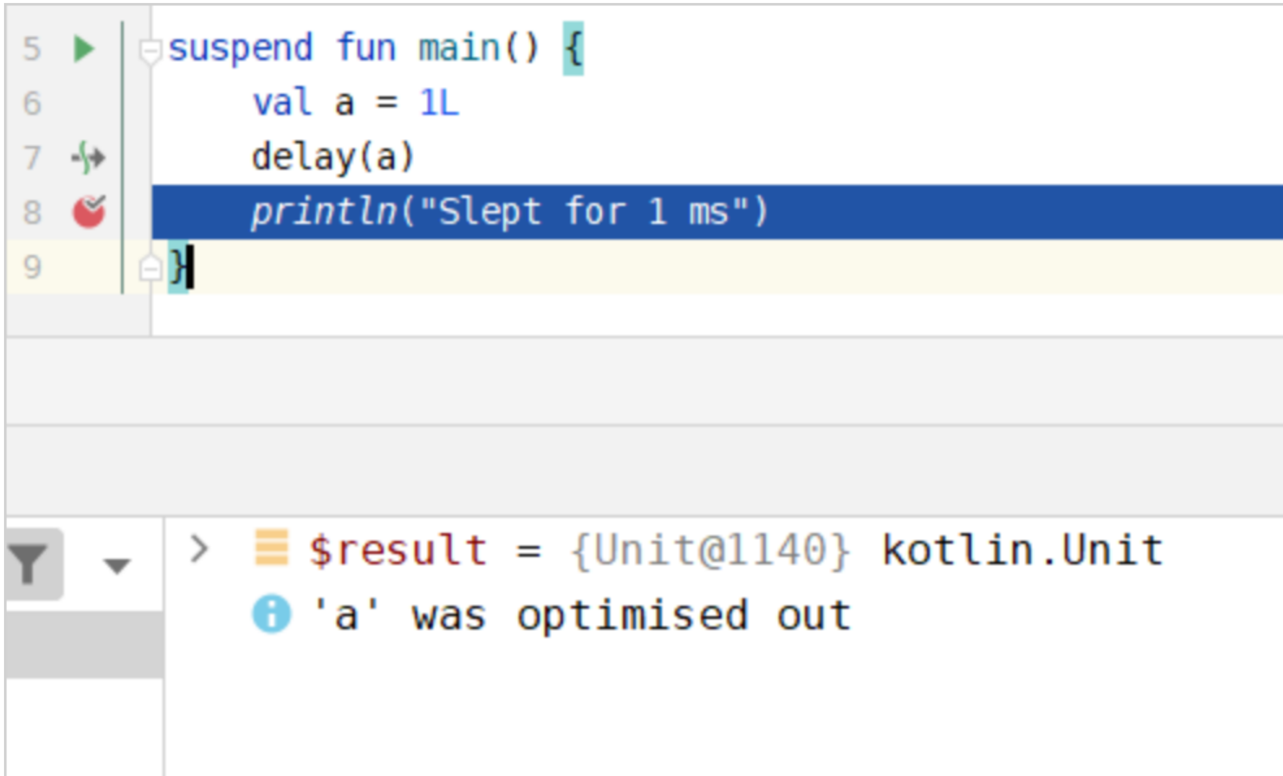
Now the flow emits the second value.



Debug the coroutine

Optimized-out variables

If you use suspend functions, in the debugger, you might see the "was optimized out" text next to a variable's name:



Variable "a" was optimized out

This text means that the variable's lifetime was decreased, and the variable doesn't exist anymore. It is difficult to debug code with optimized variables because you don't see their values. You can disable this behavior with the -Xdebug compiler option.

> Never use this flag in production: -Xdebug can cause memory leaks.

Add a concurrently running coroutine

1. Open the Main.kt file in src/main/kotlin.

2. Enhance the code to run the emitter and collector concurrently:

   - Add a call to the buffer() function to run the emitter and collector concurrently. buffer() stores emitted values and runs the flow collector in a separate coroutine.

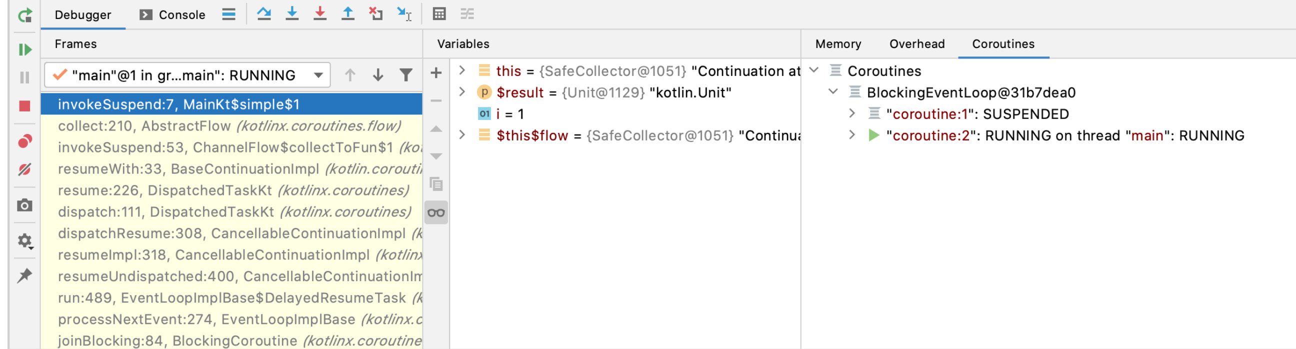
```
fun main() = runBlocking<Unit> {
    simple()
        .buffer()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

3. Build the code by clicking Build Project.

Debug a Kotlin flow with two coroutines

1. Set a new breakpoint at println(value).

2. Run the code in debug mode by clicking Debug next to the run configuration at the top of the screen.

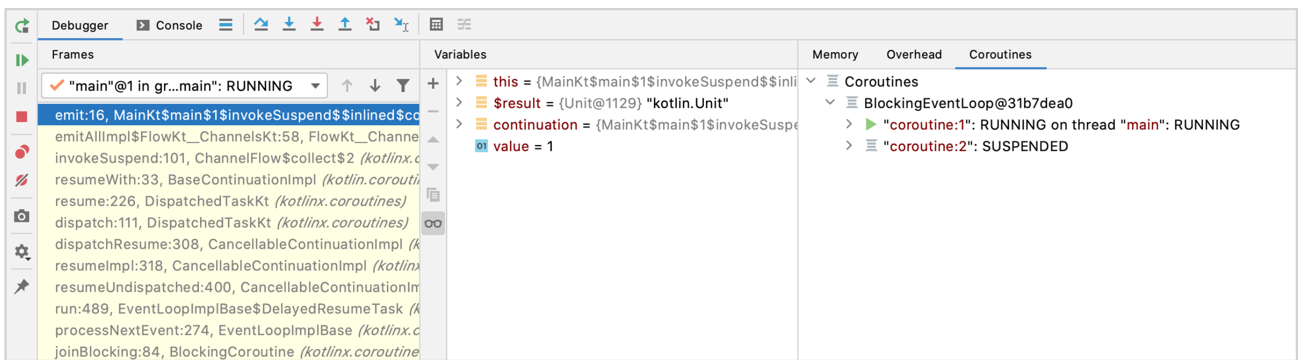


Build a console application

The Debug tool window appears.

In the Coroutines tab, you can see that there are two coroutines running concurrently. The flow collector and emitter run in separate coroutines because of the buffer() function. The buffer() function buffers emitted values from the flow. The emitter coroutine has the RUNNING status, and the collector coroutine has the SUSPENDED status.

3. Resume the debugger session by clicking Resume Program in the Debug tool window.



Debugging coroutines

Now the collector coroutine has the RUNNING status, while the emitter coroutine has the SUSPENDED status.

You can dig deeper into each coroutine to debug your code.

Serialization

Serialization is the process of converting data used by an application to a format that can be transferred over a network or stored in a database or a file. In turn, deserialization is the opposite process of reading data from an external source and converting it into a runtime object. Together they are an essential part of most applications that exchange data with third parties.

Some data serialization formats, such as JSON and protocol buffers are particularly common. Being language-neutral and platform-neutral, they enable data exchange between systems written in any modern language.

In Kotlin, data serialization tools are available in a separate component, kotlinx.serialization. It consists of two main parts: the Gradle plugin –org.jetbrains.kotlin.plugin.serialization and the runtime libraries.

Libraries

kotlinx.serialization provides sets of libraries for all supported platforms – JVM, JavaScript, Native – and for various serialization formats – JSON, CBOR, protocol buffers, and others. You can find the complete list of supported serialization formats below.

All Kotlin serialization libraries belong to the org.jetbrains.kotlinx: group. Their names start with kotlinx-serialization- and have suffixes that reflect the serialization format. Examples:

* org.jetbrains.kotlinx:kotlinx-serialization-json provides JSON serialization for Kotlin projects.
* org.jetbrains.kotlinx:kotlinx-serialization-cbor provides CBOR serialization.

Platform-specific artifacts are handled automatically; you don't need to add them manually. Use the same dependencies in JVM, JS, Native, and multiplatform projects.

Note that the kotlinx.serialization libraries use their own versioning structure, which doesn't match Kotlin's versioning. Check out the releases on GitHub to find the latest versions.

Formats

kotlinx.serialization includes libraries for various serialization formats:

* JSON: kotlinx-serialization-json
* Protocol buffers: kotlinx-serialization-protobuf
* CBOR: kotlinx-serialization-cbor
* Properties: kotlinx-serialization-properties
* HOCON: kotlinx-serialization-hocon (only on JVM)

Note that all libraries except JSON serialization (kotlinx-serialization-json) are Experimental, which means their API can be changed without notice.

There are also community-maintained libraries that support more serialization formats, such as YAML or Apache Avro. For detailed information about available serialization formats, see the kotlinx.serialization documentation.

Example: JSON serialization

Let's take a look at how to serialize Kotlin objects into JSON.

Before starting, you'll need to configure your build script so that you can use Kotlin serialization tools in your project:

1. Apply the Kotlin serialization Gradle plugin org.jetbrains.kotlin.plugin.serialization (or kotlin("plugin.serialization") in the Kotlin Gradle DSL).

Kotlin

```
plugins {
    kotlin("jvm") version "1.9.0"
    kotlin("plugin.serialization") version "1.9.0"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.9.0'
    id 'org.jetbrains.kotlin.plugin.serialization' version '1.9.0'
}
```

2. Add the JSON serialization library dependency:org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.1

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.1")
}
```

Groovy

```
dependencies {
```

```
    implementation 'org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.1'
}
```

Now you're ready to use the serialization API in your code. The API is located in the the kotlinx.serialization package and its format-specific subpackages such as kotlinx.serialization.json.

First, make a class serializable by annotating it with @Serializable.

```
import kotlinx.serialization.Serializable

@Serializable
data class Data(val a: Int, val b: String)
```

You can now serialize an instance of this class by calling Json.encodeToString().

```
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import kotlinx.serialization.encodeToString

@Serializable
data class Data(val a: Int, val b: String)

fun main() {
   val json = Json.encodeToString(Data(42, "str"))
}
```

As a result, you get a string containing the state of this object in the JSON format: {"a": 42, "b": "str"}

You can also serialize object collections, such as lists, in a single call.

```
val dataList = listOf(Data(42, "str"), Data(12, "test"))
val jsonList = Json.encodeToString(dataList)
```

To deserialize an object from JSON, use the decodeFromString() function:

```
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import kotlinx.serialization.decodeFromString

@Serializable
data class Data(val a: Int, val b: String)

fun main() {
   val obj = Json.decodeFromString<Data>("""{"a":42, "b": "str"}""")
}
```

For more information about serialization in Kotlin, see the Kotlin Serialization Guide.

Lincheck guide

Lincheck is a practical and user-friendly framework for testing concurrent algorithms on the JVM. It provides a simple and declarative way to write concurrent tests.

With the Lincheck framework, instead of describing how to perform tests, you can specify what to test by declaring all the operations to examine and the required correctness property. As a result, a typical concurrent Lincheck test contains only about 15 lines.

When given a list of operations, Lincheck automatically:

- Generates a set of random concurrent scenarios.
- Examines them using either stress-testing or bounded model checking.
- Verifies that the results of each invocation satisfy the required correctness property (linearizability is the default one).

Add Lincheck to your project

To enable the Lincheck support, include the corresponding repository and dependency to the Gradle configuration. In your build.gradle(.kts) file, add the following:

Kotlin

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation("org.jetbrains.kotlinx:lincheck:2.21")
}
```

Groovy

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation "org.jetbrains.kotlinx:lincheck:2.21"
}
```

Explore Lincheck

This guide will help you get in touch with the framework and try the most useful features with examples. Learn the Lincheck features step-by-step:

1. Write your first test with Lincheck
2. Choose your testing strategy
3. Configure operation arguments
4. Consider popular algorithm constraints
5. Check the algorithm for non-blocking progress guarantees
6. Define sequential specification of the algorithm

Additional references

- "How we test concurrent algorithms in Kotlin Coroutines" by Nikita Koval: Video. KotlinConf 2023
- "Lincheck: Testing concurrency on the JVM" workshop by Maria Sokolova: Part 1, Part 2. Hydra 2021

Write your first test with Lincheck

This tutorial demonstrates how to write your first Lincheck test, set up the Lincheck framework, and use its basic API. You will create a new IntelliJ IDEA project with an incorrect concurrent counter implementation and write a test for it, finding and analyzing the bug afterward.

Create a project

Open an existing Kotlin project in IntelliJ IDEA or create a new one. When creating a project, use the Gradle build system.

Add required dependencies

1. Open the build.gradle(.kts) file and make sure that mavenCentral() is added to the repository list.
2. Add the following dependencies to the Gradle configuration:

Kotlin

```kotlin
repositories {
    mavenCentral()
}

dependencies {
    // Lincheck dependency
    testImplementation("org.jetbrains.kotlinx:lincheck:2.21")
    // This dependency allows you to work with kotlin.test and JUnit:
    testImplementation("junit:junit:4.13")
}
```

Groovy

```groovy
repositories {
    mavenCentral()
}

dependencies {
    // Lincheck dependency
    testImplementation "org.jetbrains.kotlinx:lincheck:2.21"
    // This dependency allows you to work with kotlin.test and JUnit:
    testImplementation "junit:junit:4.13"
}
```

Write a concurrent counter and run the test

1. In the src/test/kotlin directory, create a BasicCounterTest.kt file and add the following code with a buggy concurrent counter and a Lincheck test for it:

```kotlin
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}

class BasicCounterTest {
    private val c = Counter() // Initial state

    // Operations on the Counter
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test // JUnit
    fun stressTest() = StressOptions().check(this::class) // The magic button
}
```

This Lincheck test automatically:

- Generates several random concurrent scenarios with the specified inc() and get() operations.
- Performs a lot of invocations for each of the generated scenarios.
- Verifies that each invocation result is correct.

2. Run the test above, and you will see the following error:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc(): 1 | inc(): 1 |
```

```
| ------------------- |
```

Here, Lincheck found an execution that violates the counter atomicity – two concurrent increments ended with the same result 1. It means that one increment has been lost, and the behavior of the counter is incorrect.

Trace the invalid execution

Besides showing invalid execution results, Lincheck can also provide an interleaving that leads to the error. This feature is accessible with the model checking testing strategy, which examines numerous executions with a bounded number of context switches.

1. To switch the testing strategy, replace the options type from StressOptions() to ModelCheckingOptions(). The updated BasicCounterTest class will look like this:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*

class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}

class BasicCounterTest {
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

2. Run the test again. You will get the execution trace that leads to incorrect results:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |

The following interleaving leads to the error:
| ------------------------------------------------------------------- |
| Thread 1 |                         Thread  2                        |
| ------------------------------------------------------------------- |
|          | inc()                                                    |
|          |   inc(): 1 at BasicCounterTest.inc(BasicCounterTest.kt:18) |
|          |     value.READ: 0 at Counter.inc(BasicCounterTest.kt:10)   |
|          |     switch                                               |
| inc(): 1 |                                                          |
|          |     value.WRITE(1) at Counter.inc(BasicCounterTest.kt:10)  |
|          |     value.READ: 1 at Counter.inc(BasicCounterTest.kt:10)   |
|          |   result: 1                                              |
| ------------------------------------------------------------------- |
```

According to the trace, the following events have occurred:

- T2: The second thread starts the inc() operation, reading the current counter value (value.READ: 0) and pausing.
- T1: The first thread executes inc(), which returns 1, and finishes.
- T2: The second thread resumes and increments the previously obtained counter value, incorrectly updating the counter to 1.

Get the full code.

Test the Java standard library

Let's now find a bug in the standard Java's ConcurrentLinkedDeque class. The Lincheck test below finds a race between removing and adding an element to the head of the deque:

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*
import java.util.concurrent.*

class ConcurrentDequeTest {
    private val deque = ConcurrentLinkedDeque<Int>()

    @Operation
    fun addFirst(e: Int) = deque.addFirst(e)

    @Operation
    fun addLast(e: Int) = deque.addLast(e)

    @Operation
    fun pollFirst() = deque.pollFirst()

    @Operation
    fun pollLast() = deque.pollLast()

    @Operation
    fun peekFirst() = deque.peekFirst()

    @Operation
    fun peekLast() = deque.peekLast()

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

Run modelCheckingTest(). The test will fail with the following output:

```
= Invalid execution results =
| ---------------------------------------- |
|      Thread 1     |       Thread 2       |
| ---------------------------------------- |
| addLast(22): void |                      |
| ---------------------------------------- |
| pollFirst(): 22   | addFirst(8): void    |
|                   | peekLast(): 22 [-,1] |
| ---------------------------------------- |

---
All operations above the horizontal line | ----- | happen before those below the line
---
Values in "[..]" brackets indicate the number of completed operations
in each of the parallel threads seen at the beginning of the current operation
---

The following interleaving leads to the error:
| ----------------------------------------------------------------------------------------------------------------------------
---- |
|                                              Thread 1                                                       |       Thread 2
|
| ----------------------------------------------------------------------------------------------------------------------------
---- |
| pollFirst()                                                                                                 |
|
|   pollFirst(): 22 at ConcurrentDequeTest.pollFirst(ConcurrentDequeTest.kt:17)                               |
|
|     first(): Node@1 at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:915)                      |
|
|     item.READ: null at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:917)                      |
|
|     next.READ: Node@2 at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:925)                    |
|
|     item.READ: 22 at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:917)                        |
|
|     prev.READ: null at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:919)                      |
|
|     switch                                                                                                  |
|
```

```
|                                                                                                        | addFirst(8): void
|
|                                                                                                        | peekLast(): 22
|
|     compareAndSet(Node@2,22,null): true at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:920) |
|
|     unlink(Node@2) at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:921)                  |
|
|   result: 22                                                                                           |
|
| ------------------------------------------------------------------------------------------------------------------------------
---- |
```

Get the full code.

Next step

Choose your testing strategy and configure test execution.

See also

- How to generate operation arguments
- Popular algorithm constraints
- Checking for non-blocking progress guarantees
- Define sequential specification of the algorithm

Stress testing and model checking

Lincheck offers two testing strategies: stress testing and model checking. Learn what happens under the hood of both approaches using the Counter we coded in the BasicCounterTest.kt file in the previous step:

```
class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}
```

Stress testing

Write a stress test

Create a concurrent stress test for the Counter, following these steps:

1. Create the CounterTest class.
2. In this class, add the field c of the Counter type, creating an instance in the constructor.
3. List the counter operations and mark them with the @Operation annotation, delegating their implementations to c.
4. Specify the stress testing strategy using StressOptions().
5. Invoke the StressOptions.check() function to run the test.

The resulting code will look like this:

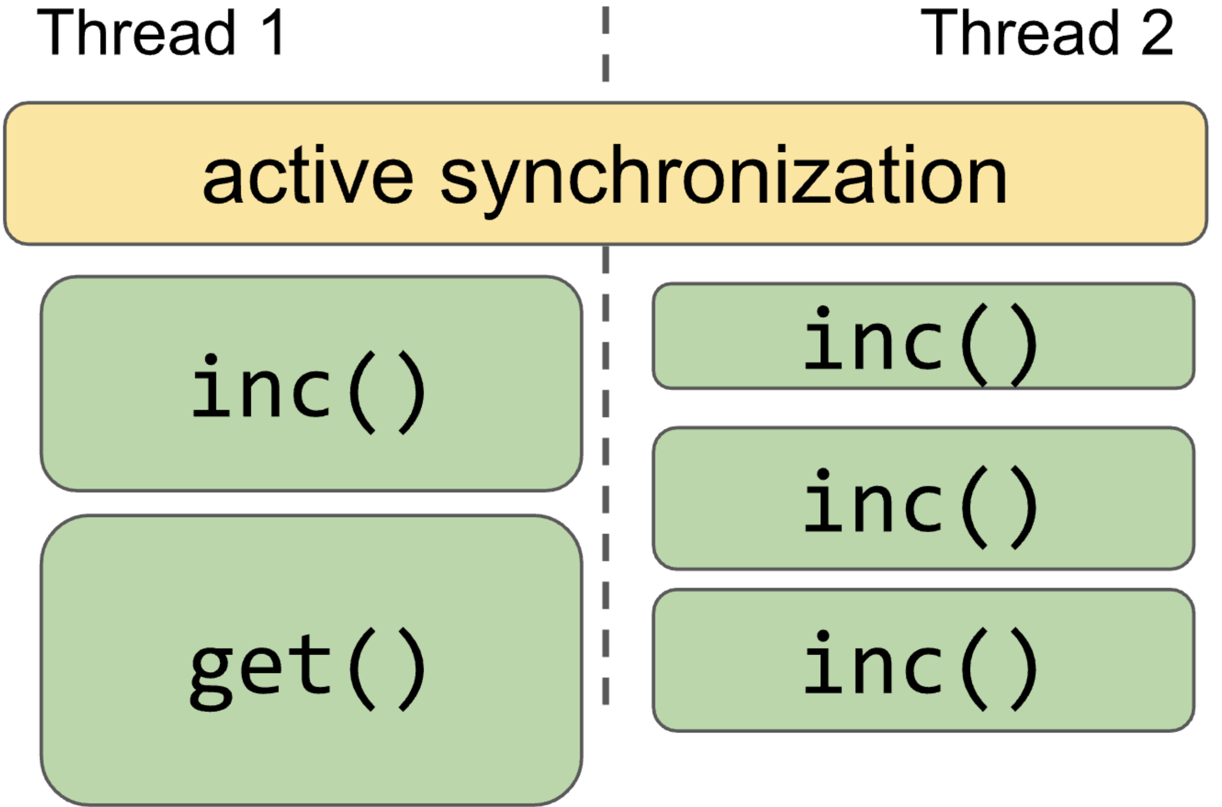
```
import org.jetbrains.kotlinx.lincheck.annotations.*
```

```kotlin
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class CounterTest {
    private val c = Counter() // Initial state

    // Operations on the Counter
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test // Run the test
    fun stressTest() = StressOptions().check(this::class)
}
```

How stress testing works

At first, Lincheck generates a set of concurrent scenarios using the operations marked with @Operation. Then it launches native threads, synchronizing them at the beginning to guarantee that operations start simultaneously. Finally, Lincheck executes each scenario on these native threads multiple times, expecting to hit an interleaving that produces incorrect results.

The figure below shows a high-level scheme of how Lincheck may execute generated scenarios:



Stress execution of the Counter

Model checking

The main concern regarding stress testing is that you may spend hours trying to understand how to reproduce the found bug. To help you with that, Lincheck supports bounded model checking, which automatically provides an interleaving for reproducing bugs.

A model checking test is constructed the same way as the stress test. Just replace the StressOptions() that specify the testing strategy with

ModelCheckingOptions().

Write a model checking test

To change the stress testing strategy to model checking, replace StressOptions() with ModelCheckingOptions() in your test:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*

class CounterTest {
    private val c = Counter() // Initial state

    // Operations on the Counter
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test // Run the test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

> To use model checking strategy for Java 9 and later, add the following JVM properties:
>
> ```
> --add-opens java.base/jdk.internal.misc=ALL-UNNAMED
> --add-exports java.base/jdk.internal.util=ALL-UNNAMED
> ```
>
> They are required if the testing code uses classes from the java.util package since some of them use jdk.internal.misc.Unsafe or similar internal classes under the hood. If you use Gradle, add the following lines to build.gradle.kts:
>
> tasks.withType<Test> { jvmArgs("--add-opens", "java.base/jdk.internal.misc=ALL-UNNAMED", "--add-exports", "java.base/jdk.internal.util=ALL-UNNAMED", "--add-exports", "java.base/sun.security.action=ALL-UNNAMED") }

How model checking works

Most bugs in complicated concurrent algorithms can be reproduced with classic interleavings, switching the execution from one thread to another. Besides, model checkers for weak memory models are very complicated, so Lincheck uses a bounded model checking under the sequential consistency memory model.

In short, Lincheck analyzes all interleavings, starting with one context switch, then two, continuing the process until the specified number of interleaving is examined. This strategy allows finding an incorrect schedule with the lowest possible number of context switches, making further bug investigation easier.

To control the execution, Lincheck inserts special switch points into the testing code. These points identify where a context switch can be performed. Essentially, these are shared memory accesses, such as field and array element reads or updates in the JVM, as well as wait/notify and park/unpark calls. To insert a switch point, Lincheck transforms the testing code on the fly using the ASM framework, adding internal function invocations to the existing code.

As the model checking strategy controls the execution, Lincheck can provide the trace that leads to the invalid interleaving, which is extremely helpful in practice. You can see the example of trace for the incorrect execution of the Counter in the Write your first test with Lincheck tutorial.

Which testing strategy is better?

The model checking strategy is preferable for finding bugs under the sequentially consistent memory model since it ensures better coverage and provides a failing execution trace if an error is found.

Although stress testing doesn't guarantee any coverage, checking algorithms for bugs introduced by low-level effects, such as a missed volatile modifier, is still helpful. Stress testing is also a great help in discovering rare bugs that require many context switches to reproduce, and it's impossible to analyze them all due to the current restrictions in the model checking strategy.

Configure the testing strategy

To configure the testing strategy, set options in the <TestingMode>Options class.

1. Set the options for scenario generation and execution for the CounterTest:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class CounterTest {
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test
    fun stressTest() = StressOptions() // Stress testing options:
        .actorsBefore(2) // Number of operations before the parallel part
        .threads(2) // Number of threads in the parallel part
        .actorsPerThread(2) // Number of operations in each thread of the parallel part
        .actorsAfter(1) // Number of operations after the parallel part
        .iterations(100) // Generate 100 random concurrent scenarios
        .invocationsPerIteration(1000) // Run each generated scenario 1000 times
        .check(this::class) // Run the test
}
```

2. Run stressTest() again, Lincheck will generate scenarios similar to the one below:

```
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc()    |          |
| inc()    |          |
| ------------------- |
| get()    | inc()    |
| inc()    | get()    |
| ------------------- |
| inc()    |          |
| ------------------- |
```

Here, there are two operations before the parallel part, two threads for each of the two operations, followed after that by a single operation in the end.

You can configure your model checking tests in the same way.

Scenario minimization

You may already have noticed that detected errors are usually represented with a scenario smaller than the specified in the test configuration. Lincheck tries to minimize the error, actively removing an operation while it's possible to keep the test from failing.

Here's the minimized scenario for the counter test above:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc()    | inc()    |
| ------------------- |
```

As it's easier to analyze smaller scenarios, scenario minimization is enabled by default. To disable this feature, add minimizeFailedScenario(false) to the [Stress, ModelChecking]Options configuration.

Logging data structure states

Another useful feature for debugging is state logging. When analyzing an interleaving that leads to an error, you usually draw the data structure changes on a sheet of paper, changing the state after each event. To automize this procedure, you can provide a special method that returns a String representation of the data structure, so Lincheck prints the state representation after each event in the interleaving that modifies the data structure.

For this, define a method that doesn't take arguments and is marked with the @StateRepresentation annotation. The method should be thread-safe, non-blocking,

and never modify the data structure.

1. In the Counter example, the String representation is simply the value of the counter. Thus, to print the counter states in the trace, add the stateRepresentation() function to the CounterTest:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.Test

class CounterTest {
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @StateRepresentation
    fun stateRepresentation() = c.get().toString()

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

2. Run the modelCheckingTest() now and check the states of the Counter printed at the switch points that modify the counter state (they start with STATE:):

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| STATE: 0            |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |
| STATE: 1            |
| ------------------- |

The following interleaving leads to the error:
| -------------------------------------------------------------------- |
| Thread 1 |                         Thread 2                          |
| -------------------------------------------------------------------- |
|          | inc()                                                     |
|          |   inc(): 1 at CounterTest.inc(CounterTest.kt:10)          |
|          |     value.READ: 0 at Counter.inc(BasicCounterTest.kt:10)  |
|          |     switch                                                |
| inc(): 1 |                                                           |
| STATE: 1 |                                                           |
|          |     value.WRITE(1) at Counter.inc(BasicCounterTest.kt:10) |
|          |     STATE: 1                                              |
|          |     value.READ: 1 at Counter.inc(BasicCounterTest.kt:10)  |
|          |   result: 1                                               |
| -------------------------------------------------------------------- |
```

In case of stress testing, Lincheck prints the state representation right before and after the parallel part of the scenario, as well as at the end.

- Get the full code of these examples
- See more test examples

Next step

Learn how to configure arguments passed to the operations and when it can be useful.

Operation arguments

In this tutorial, you'll learn how to configure operation arguments.

Consider this straightforward MultiMap implementation below. It's based on the ConcurrentHashMap, internally storing a list of values:

```
import java.util.concurrent.*

class MultiMap<K, V> {
    private val map = ConcurrentHashMap<K, List<V>>()

    // Maintains a list of values
    // associated with the specified key.
    fun add(key: K, value: V) {
        val list = map[key]
        if (list == null) {
            map[key] = listOf(value)
        } else {
            map[key] = list + value
        }
    }

    fun get(key: K): List<V> = map[key] ?: emptyList()
}
```

Is this MultiMap implementation linearizable? If not, an incorrect interleaving is more likely to be detected when accessing a small range of keys, thus, increasing the possibility of processing the same key concurrently.

For this, configure the generator for a key: Int parameter:

1. Declare the @Param annotation.
2. Specify the integer generator class: @Param(gen = IntGen::class). Lincheck supports random parameter generators for almost all primitives and strings out of the box.
3. Define the range of values generated with the string configuration @Param(conf = "1:2").
4. Specify the parameter configuration name (@Param(name = "key")) to share it for several operations.

   Below is the stress test for MultiMap that generates keys for add(key, value) and get(key) operations in the range of [1..2]:

```
import java.util.concurrent.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.paramgen.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*

class MultiMap<K, V> {
    private val map = ConcurrentHashMap<K, List<V>>()

    // Maintains a list of values
    // associated with the specified key.
    fun add(key: K, value: V) {
        val list = map[key]
        if (list == null) {
            map[key] = listOf(value)
        } else {
            map[key] = list + value
        }
    }

    fun get(key: K): List<V> = map[key] ?: emptyList()
}

@Param(name = "key", gen = IntGen::class, conf = "1:2")
class MultiMapTest {
    private val map = MultiMap<Int, Int>()

    @Operation
    fun add(@Param(name = "key") key: Int, value: Int) = map.add(key, value)

    @Operation
    fun get(@Param(name = "key") key: Int) = map.get(key)

    @Test
    fun stressTest() = StressOptions().check(this::class)
```

```
    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

5. Run the stressTest() and see the following output:

```
= Invalid execution results =
| --------------------------------- |
|    Thread 1     |     Thread 2     |
| --------------------------------- |
| add(2, 0): void | add(2, -1): void |
| --------------------------------- |
| get(2): [0]     |                  |
| --------------------------------- |
```

6. Finally, run modelCheckingTest(). It fails with the following output:

```
= Invalid execution results =
| --------------------------------- |
|    Thread 1     |     Thread 2     |
| --------------------------------- |
| add(2, 0): void | add(2, -1): void |
| --------------------------------- |
| get(2): [-1]    |                  |
| --------------------------------- |

---
All operations above the horizontal line | ----- | happen before those below the line
---

The following interleaving leads to the error:
| --------------------------------------------------------------------- |
|    Thread 1     |                      Thread 2                       |
| --------------------------------------------------------------------- |
|                 | add(2, -1)                                          |
|                 |   add(2,-1) at MultiMapTest.add(MultiMap.kt:31)     |
|                 |     get(2): null at MultiMap.add(MultiMap.kt:15)    |
|                 |     switch                                          |
| add(2, 0): void |                                                     |
|                 |     put(2,[-1]): [0] at MultiMap.add(MultiMap.kt:17) |
|                 |   result: void                                      |
| --------------------------------------------------------------------- |
```

Due to the small range of keys, Lincheck quickly reveals the race: when two values are being added concurrently by the same key, one of the values may be overwritten and lost.

Get the full code.

Next step

Learn how to test data structures that set access constraints on the execution, such as single-producer single-consumer queues.

Data structure constraints

Some data structures may require a part of operations not to be executed concurrently, such as single-producer single-consumer queues. Lincheck provides out-of-the-box support for such contracts, generating concurrent scenarios according to the restrictions.

Consider the single-consumer queue from the JCTools library. Let's write a test to check correctness of its poll(), peek(), and offer(x) operations.

In your build.gradle(.kts) file, add the JCTools dependency:

Kotlin

```
dependencies {
    // jctools dependency
    testImplementation("org.jctools:jctools-core:3.3.0")
```

```
}
```

Groovy

```
dependencies {
    // jctools dependency
    testImplementation "org.jctools:jctools-core:3.3.0"
}
```

To meet the single-consumer restriction, ensure that all poll() and peek() consuming operations are called from a single thread. For that, we can set the nonParallelGroup parameter of the corresponding @Operation annotations to the same value, e.g. "consumers".

Here is the resulting test:

```
import org.jctools.queues.atomic.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class MPSCQueueTest {
    private val queue = MpscLinkedAtomicQueue<Int>()

    @Operation
    fun offer(x: Int) = queue.offer(x)

    @Operation(nonParallelGroup = "consumers")
    fun poll(): Int? = queue.poll()

    @Operation(nonParallelGroup = "consumers")
    fun peek(): Int? = queue.peek()

    @Test
    fun stressTest() = StressOptions().check(this::class)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

Here is an example of the scenario generated for this test:

```
= Iteration 15 / 100 =
| --------------------- |
| Thread 1  | Thread 2  |
| --------------------- |
| poll()    |           |
| poll()    |           |
| peek()    |           |
| peek()    |           |
| peek()    |           |
| --------------------- |
| offer(-1) | offer(0)  |
| offer(0)  | offer(-1) |
| peek()    | offer(-1) |
| offer(1)  | offer(1)  |
| peek()    | offer(1)  |
| --------------------- |
| peek()    |           |
| offer(-2) |           |
| offer(-2) |           |
| offer(2)  |           |
| offer(-2) |           |
| --------------------- |
```

Note that all consuming poll() and peek() invocations are performed from a single thread, thus satisfying the "single-consumer" restriction.

Get the full code.

Next step

Learn how to check your algorithm for progress guarantees with the model checking strategy.

Progress guarantees

Many concurrent algorithms provide non-blocking progress guarantees, such as lock-freedom and wait-freedom. As they are usually non-trivial, it's easy to add a bug that blocks the algorithm. Lincheck can help you find liveness bugs using the model checking strategy.

To check the progress guarantee of the algorithm, enable the checkObstructionFreedom option in ModelCheckingOptions():

```
ModelCheckingOptions().checkObstructionFreedom()
```

Create a ConcurrentMapTest.kt file. Then add the following test to detect that ConcurrentHashMap::put(key: K, value: V) from the Java standard library is a blocking operation:

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*
import java.util.concurrent.*

class ConcurrentHashMapTest {
    private val map = ConcurrentHashMap<Int, Int>()

    @Operation
    fun put(key: Int, value: Int) = map.put(key, value)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .actorsBefore(1) // To init the HashMap
        .actorsPerThread(1)
        .actorsAfter(0)
        .minimizeFailedScenario(false)
        .checkObstructionFreedom()
        .check(this::class)
}
```

Run the modelCheckingTest(). You should get the following result:

```
= Obstruction-freedom is required but a lock has been found =
| --------------------- |
|  Thread 1  | Thread 2  |
| --------------------- |
| put(1, -1) |           |
| --------------------- |
| put(2, -2) | put(3, 2) |
| --------------------- |

---
All operations above the horizontal line | ----- | happen before those below the line
---

The following interleaving leads to the error:
| -----------------------------------------------------------------------------------------------------------------
------------------------------------------------------ |
|                                          Thread 1                                          |
Thread 2                                         |
| -----------------------------------------------------------------------------------------------------------------
------------------------------------------------------ |
|                                                                                            | put(3, 2)
|
|                                                                                            |   put(3,2) at
ConcurrentHashMapTest.put(ConcurrentMapTest.kt:11)                          |
|                                                                                            |     putVal(3,2,false) at
ConcurrentHashMap.put(ConcurrentHashMap.java:1006)              |
|                                                                                            |       table.READ: Node[]@1 at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1014)       |
|                                                                                            |       tabAt(Node[]@1,0): Node@1 at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1018) |
|                                                                                            |       MONITORENTER at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1031)               |
```

```
|                                                                                           |       tabAt(Node[]@1,0): Node@1 at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1032) |
|                                                                                           |       next.READ: null at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1046)           |
|                                                                                           |       switch
|
| put(2, -2)                                                                                |
|
|   put(2,-2) at ConcurrentHashMapTest.put(ConcurrentMapTest.kt:11)                         |
|
|     putVal(2,-2,false) at ConcurrentHashMap.put(ConcurrentHashMap.java:1006)              |
|
|       table.READ: Node[]@1 at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1014)       |
|
|       tabAt(Node[]@1,0): Node@1 at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1018) |
|
|       MONITORENTER at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1031)               |
|
| ---------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------ |
```

Now let's add a test for the non-blocking ConcurrentSkipListMap<K, V>, expecting the test to pass successfully:

```
class ConcurrentSkipListMapTest {
    private val map = ConcurrentSkipListMap<Int, Int>()

    @Operation
    fun put(key: Int, value: Int) = map.put(key, value)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .checkObstructionFreedom()
        .check(this::class)
}
```

The common non-blocking progress guarantees are (from strongest to weakest):

- wait-freedom, when each operation is completed in a bounded number of steps no matter what other threads do.
- lock-freedom, which guarantees system-wide progress so that at least one operation is completed in a bounded number of steps while a particular operation may be stuck.
- obstruction-freedom, when any operation is completed in a bounded number of steps if all the other threads pause.

At the moment, Lincheck supports only the obstruction-freedom progress guarantees. However, most real-life liveness bugs add unexpected blocking code, so the obstruction-freedom check will also help with lock-free and wait-free algorithms.

- Get the full code of the example.
- See another example where the Michael-Scott queue implementation is tested for progress guarantees.

Next step

Learn how to specify the sequential specification of the testing algorithm explicitly, improving the Lincheck tests robustness.

Sequential specification

To be sure that the algorithm provides correct sequential behavior, you can define its sequential specification by writing a straightforward sequential implementation of the testing data structure.

This feature also allows you to write a single test instead of two separate sequential and concurrent tests.

To provide a sequential specification of the algorithm for verification:

1. Implement a sequential version of all the testing methods.

2. Pass the class with sequential implementation to the sequentialSpecification() option:

```
StressOptions().sequentialSpecification(SequentialQueue::class)
```

For example, here is the test to check correctness of j.u.c.ConcurrentLinkedQueue from the Java standard library.

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*
import java.util.*
import java.util.concurrent.*

class ConcurrentLinkedQueueTest {
    private val s = ConcurrentLinkedQueue<Int>()

    @Operation
    fun add(value: Int) = s.add(value)

    @Operation
    fun poll(): Int? = s.poll()

    @Test
    fun stressTest() = StressOptions()
        .sequentialSpecification(SequentialQueue::class.java)
        .check(this::class)
}

class SequentialQueue {
    private val s = LinkedList<Int>()

    fun add(x: Int) = s.add(x)
    fun poll(): Int? = s.poll()
}
```

Get the full code of the examples.

Keywords and operators

Hard keywords

The following tokens are always interpreted as keywords and cannot be used as identifiers:

- as
  - is used for type casts.
  - specifies an alias for an import
- as? is used for safe type casts.
- break terminates the execution of a loop.
- class declares a class.
- continue proceeds to the next step of the nearest enclosing loop.
- do begins a do/while loop (a loop with a postcondition).
- else defines the branch of an if expression that is executed when the condition is false.

- false specifies the 'false' value of the Boolean type.
- for begins a for loop.
- fun declares a function.
- if begins an if expression.
- in
  - specifies the object being iterated in a for loop.
  - is used as an infix operator to check that a value belongs to a range, a collection, or another entity that defines a 'contains' method.
  - is used in when expressions for the same purpose.
  - marks a type parameter as contravariant.
- !in
  - is used as an operator to check that a value does NOT belong to a range, a collection, or another entity that defines a 'contains' method.
  - is used in when expressions for the same purpose.
- interface declares an interface.
- is
  - checks that a value has a certain type.
  - is used in when expressions for the same purpose.
- !is
  - checks that a value does NOT have a certain type.
  - is used in when expressions for the same purpose.
- null is a constant representing an object reference that doesn't point to any object.
- object declares a class and its instance at the same time.
- package specifies the package for the current file.
- return returns from the nearest enclosing function or anonymous function.
- super
  - refers to the superclass implementation of a method or property.
  - calls the superclass constructor from a secondary constructor.
- this
  - refers to the current receiver.
  - calls another constructor of the same class from a secondary constructor.
- throw throws an exception.
- true specifies the 'true' value of the Boolean type.
- try begins an exception-handling block.
- typealias declares a type alias.
- typeof is reserved for future use.
- val declares a read-only property or local variable.
- var declares a mutable property or local variable.
- when begins a when expression (executes one of the given branches).

- while begins a while loop (a loop with a precondition).

Soft keywords

The following tokens act as keywords in the context in which they are applicable, and they can be used as identifiers in other contexts:

- by
  - delegates the implementation of an interface to another object.
  - delegates the implementation of the accessors for a property to another object.
- catch begins a block that handles a specific exception type.
- constructor declares a primary or secondary constructor.
- delegate is used as an annotation use-site target.
- dynamic references a dynamic type in Kotlin/JS code.
- field is used as an annotation use-site target.
- file is used as an annotation use-site target.
- finally begins a block that is always executed when a try block exits.
- get
  - declares the getter of a property.
  - is used as an annotation use-site target.
- import imports a declaration from another package into the current file.
- init begins an initializer block.
- param is used as an annotation use-site target.
- property is used as an annotation use-site target.
- receiveris used as an annotation use-site target.
- set
  - declares the setter of a property.
  - is used as an annotation use-site target.
- setparam is used as an annotation use-site target.
- value with the class keyword declares an inline class.
- where specifies the constraints for a generic type parameter.

Modifier keywords

The following tokens act as keywords in modifier lists of declarations, and they can be used as identifiers in other contexts:

- abstract marks a class or member as abstract.
- actual denotes a platform-specific implementation in multiplatform projects.
- annotation declares an annotation class.
- companion declares a companion object.
- const marks a property as a compile-time constant.
- crossinline forbids non-local returns in a lambda passed to an inline function.

- data instructs the compiler to generate canonical members for a class.
- enum declares an enumeration.
- expect marks a declaration as platform-specific, expecting an implementation in platform modules.
- external marks a declaration as implemented outside of Kotlin (accessible through JNI or in JavaScript).
- final forbids overriding a member.
- infix allows calling a function using infix notation.
- inline tells the compiler to inline a function and the lambdas passed to it at the call site.
- inner allows referring to an outer class instance from a nested class.
- internal marks a declaration as visible in the current module.
- lateinit allows initializing a non-null property outside of a constructor.
- noinline turns off inlining of a lambda passed to an inline function.
- open allows subclassing a class or overriding a member.
- operator marks a function as overloading an operator or implementing a convention.
- out marks a type parameter as covariant.
- override marks a member as an override of a superclass member.
- private marks a declaration as visible in the current class or file.
- protected marks a declaration as visible in the current class and its subclasses.
- public marks a declaration as visible anywhere.
- reified marks a type parameter of an inline function as accessible at runtime.
- sealed declares a sealed class (a class with restricted subclassing).
- suspend marks a function or lambda as suspending (usable as a coroutine).
- tailrec marks a function as tail-recursive (allowing the compiler to replace recursion with iteration).
- vararg allows passing a variable number of arguments for a parameter.

Special identifiers

The following identifiers are defined by the compiler in specific contexts, and they can be used as regular identifiers in other contexts:

- field is used inside a property accessor to refer to the backing field of the property.
- it is used inside a lambda to refer to its parameter implicitly.

Operators and special symbols

Kotlin supports the following operators and special symbols:

- +, -, *, /, % - mathematical operators
  - * is also used to pass an array to a vararg parameter.
- =
  - assignment operator.
  - is used to specify default values for parameters.
- +=, -=, *=, /=, %= - augmented assignment operators.

* ++, -- - increment and decrement operators.
* &&, ||, ! - logical 'and', 'or', 'not' operators (for bitwise operations, use the corresponding infix functions instead).
* ==, != - equality operators (translated to calls of equals() for non-primitive types).
* ===, !== - referential equality operators.
* <, >, <=, >= - comparison operators (translated to calls of compareTo() for non-primitive types).
* [,] - indexed access operator (translated to calls of get and set).
* !! asserts that an expression is non-null.
* ?. performs a safe call (calls a method or accesses a property if the receiver is non-null).
* ?: takes the right-hand value if the left-hand value is null (the elvis operator).
* :: creates a member reference or a class reference.
* .., ..< create ranges.
* : separates a name from a type in a declaration.
* ? marks a type as nullable.
* ->
  * separates the parameters and body of a lambda expression.
  * separates the parameters and return type declaration in a function type.
  * separates the condition and body of a when expression branch.
* @
  * introduces an annotation.
  * introduces or references a loop label.
  * introduces or references a lambda label.
  * references a 'this' expression from an outer scope.
  * references an outer superclass.
* ; separates multiple statements on the same line.
* $ references a variable or expression in a string template.
* _
  * substitutes an unused parameter in a lambda expression.
  * substitutes an unused parameter in a destructuring declaration.

For operator precedence, see this reference in Kotlin grammar.

Gradle

Gradle is a build system that helps to automate and manage your building process. It downloads required dependencies, packages your code, and prepares it for compilation. Learn about Gradle basics and specifics on the Gradle website.

You can set up your own project with these instructions for different platforms or pass a small step-by-step tutorial that will show you how to create a simple backend "Hello World" application in Kotlin.

> You can find information about the compatibility of Kotlin, Gradle, and Android Gradle plugin versions here.

In this chapter, you can also learn about:

- Compiler options and how to pass them.
- Incremental compilation, caches support, build reports, and the Kotlin daemon.
- Support for Gradle plugin variants.

What's next?

Learn about:

- Gradle Kotlin DSL. The Gradle Kotlin DSL is a domain specific language that you can use to write build scripts quickly and efficiently.
- Annotation processing. Kotlin supports annotation processing via the Kotlin Symbol processing API.
- Generating documentation. To generate documentation for Kotlin projects, use Dokka; please refer to the Dokka README for configuration instructions. Dokka supports mixed-language projects and can generate output in multiple formats, including standard Javadoc.
- OSGi. For OSGi support see the Kotlin OSGi page.

Get started with Gradle and Kotlin/JVM

This tutorial demonstrates how to use IntelliJ IDEA and Gradle for creating a console application.

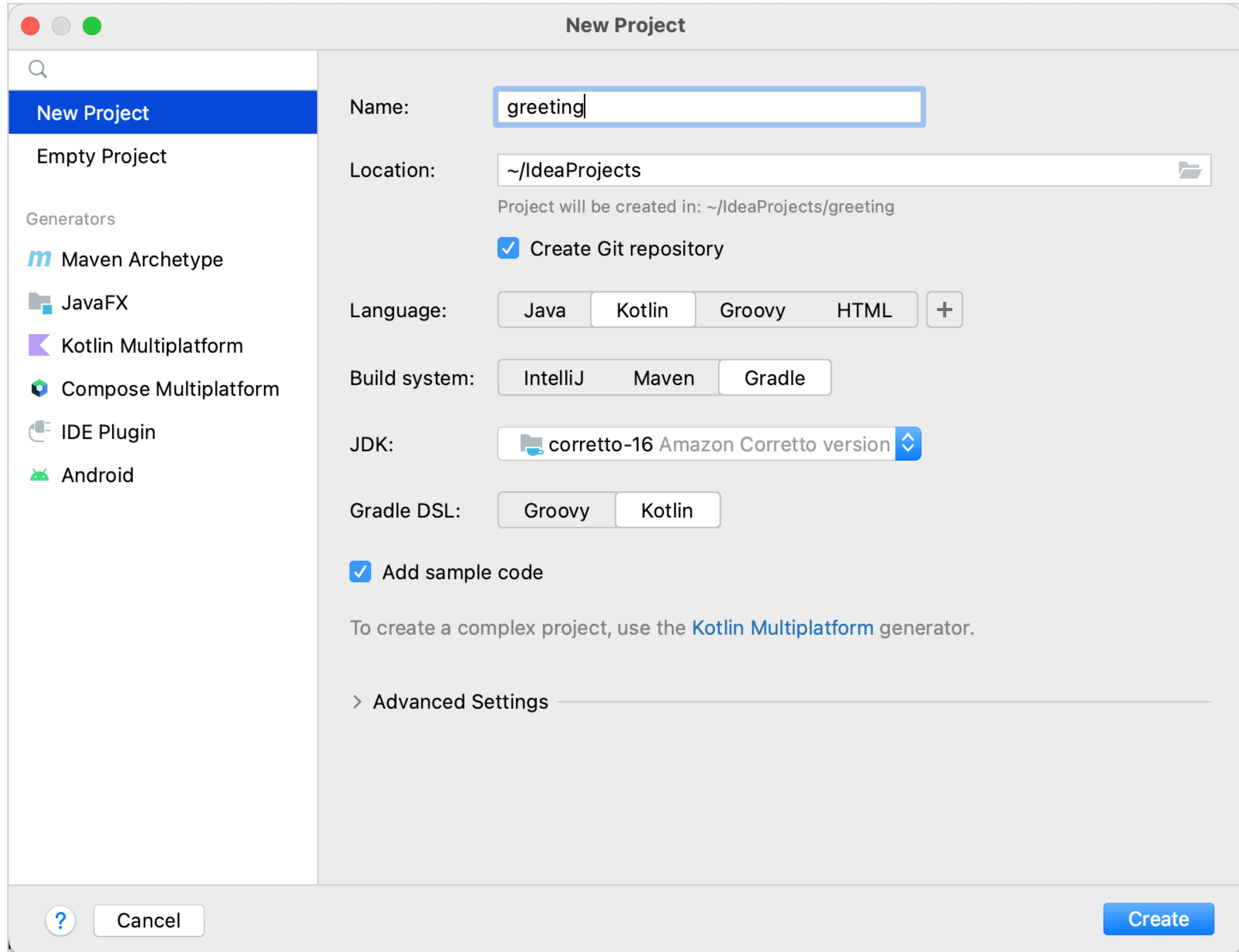
To get started, first download and install the latest version of IntelliJ IDEA.

Create a project

1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select New Project.
3. Name the new project and change its location, if necessary.

   > Select the Create Git repository checkbox to place the new project under version control. You will be able to do it later at any time.

4. From the Language list, select Kotlin.



Create a console application

5. Select the Gradle build system.

6. From the JDK list, select the JDK that you want to use in your project.

   - If the JDK is installed on your computer, but not defined in the IDE, select Add JDK and specify the path to the JDK home directory.
   - If you don't have the necessary JDK on your computer, select Download JDK.

7. From the Gradle DSL list, select Kotlin.

8. Select the Add sample code checkbox to create a file with a sample "Hello World!" application.

9. Click Create.

You have successfully created a project with Gradle.

Explore the build script

Open the build.gradle.kts file. This is the Gradle Kotlin build script, which contains Kotlin-related artifacts and other parts required for the application:

```
 // For `KotlinCompile` task below
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    kotlin("jvm") version "1.9.0" // Kotlin version to use
    application // Application plugin. Also see    below the code
}
```

```
group = "org.example" // A company name, for example, `org.jetbrains`
version = "1.0-SNAPSHOT" // Version to assign to the built artifact

repositories { // Sources of dependencies. See
    mavenCentral() // Maven Central Repository. See
}

dependencies { // All the libraries you want to use. See
    // Copy dependencies' names after you find them in a repository
    testImplementation(kotlin("test")) // The Kotlin test library
}

tasks.test { // See
    useJUnitPlatform() // JUnitPlatform for tests. See
}

kotlin { // Extension for easy setup
    jvmToolchain(17) // Target version of generated JVM bytecode. See
}

application {
    mainClass.set("MainKt") // The main class of the application
}
```

- Application plugin to add support for building CLI application in Java.
- Lean more about sources of dependencies.
- The Maven Central Repository. It can also be Google's Maven repository or your company's private repository.
- Learn more about declaring dependencies.
- Learn more about tasks.
- JUnitPlatform for tests.
- Learn more about setting up a Java toolchain.

As you can see, there are a few Kotlin-specific artifacts added to the Gradle build file:

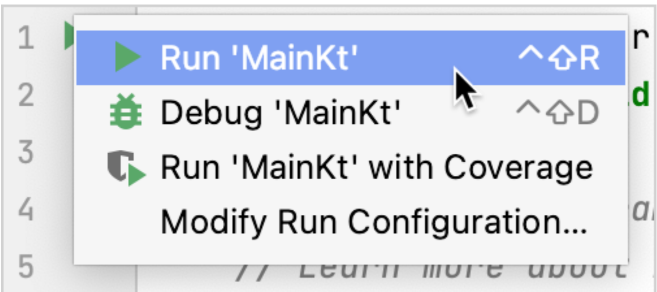
1. In the plugins block, there is the kotlin("jvm") artifact – the plugin defines the version of Kotlin to be used in the project.
2. In the dependencies section, there is testImplementation(kotlin("test")). Learn more about setting dependencies on test libraries.
3. After the dependencies section, there is the KotlinCompile task configuration block. This is where you can add extra arguments to the compiler to enable or disable various language features.

Run the application

Open the Main.kt file in src/main/kotlin.
The src directory contains Kotlin source files and resources. The Main.kt file contains sample code that will print Hello World!.

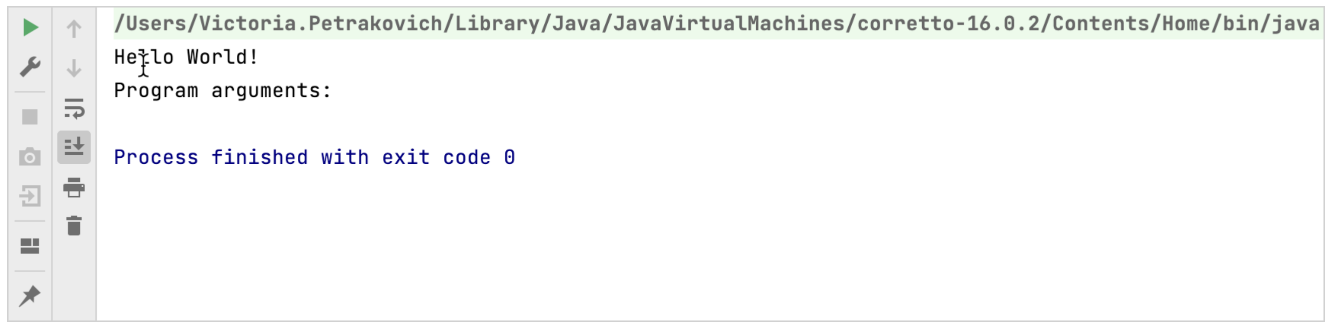
Main.kt with main fun

The easiest way to run the application is to click the green Run icon in the gutter and select Run 'MainKt'.



Running a console app

You can see the result in the Run tool window.



Kotlin run output

Congratulations! You have just run your first Kotlin application.

What's next?

Learn more about:

- Gradle build file properties.
- Targeting different platforms and setting library dependencies.
- Compiler options and how to pass them.
- Incremental compilation, caches support, build reports, and the Kotlin daemon.

Configure a Gradle project

To build a Kotlin project with Gradle, you need to add the Kotlin Gradle plugin to your build script file build.gradle(.kts) and configure the project's dependencies there.

> To learn more about the contents of a build script, visit the Explore the build script section.

Apply the plugin

To apply the Kotlin Gradle plugin, use the plugins block from the Gradle plugins DSL:

Kotlin

```
// replace `<...>` with the plugin name
plugins {
    kotlin("<...>") version "1.9.0"
}
```

Groovy

```
// replace `<...>` with the plugin name
plugins {
    id 'org.jetbrains.kotlin.<...>' version '1.9.0'
}
```

The Kotlin Gradle plugin (KGP) and Kotlin share the same version numbering.

When configuring your project, check the Kotlin Gradle plugin (KGP) compatibility with available Gradle versions. In the following table, there are the minimum and maximum fully supported versions of Gradle and Android Gradle plugin (AGP):

KGP version	Gradle min and max versions	AGP min and max versions
1.9.0	6.8.3 – 7.6.0	4.2.2 – 7.4.0
1.8.20	6.8.3 – 7.6.0	4.1.3 – 7.4.0
1.8.0	6.8.3 – 7.3.3	4.1.3 – 7.2.1
1.7.20	6.7.1 – 7.1.1	3.6.4 – 7.0.4
1.7.0	6.7.1 – 7.0.2	3.4.3 – 7.0.2
1.6.20	6.1.1 - 7.0.2	3.4.3 - 7.0.2

You can also use Gradle and AGP versions up to the latest releases, but if you do, keep in mind that you might encounter deprecation warnings or some new features might not work.

For example, the Kotlin Gradle plugin and the kotlin-multiplatform plugin 1.9.0 require the minimum Gradle version of 6.8.3 for your project to compile.

Similarly, the maximum fully supported version is 7.6.0. It doesn't have deprecated Gradle methods and properties, and supports all the current Gradle features.

Targeting the JVM

To target the JVM, apply the Kotlin JVM plugin.

Kotlin

```
plugins {
    kotlin("jvm") version "1.9.0"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.9.0"
}
```

The version should be literal in this block, and it cannot be applied from another build script.

Kotlin and Java sources

Kotlin sources and Java sources can be stored in the same directory, or they can be placed in different directories.

The default convention is to use different directories:

```
project
    - src
        - main (root)
            - kotlin
            - java
```

> Do not store Java .java files in the src/*/kotlin directory, as the .java files will not be compiled.
>
> Instead, you can use src/main/java.

The corresponding sourceSets property should be updated if you are not using the default convention:

Kotlin

```
sourceSets.main {
    java.srcDirs("src/main/myJava", "src/main/myKotlin")
}
```

Groovy

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

Check for JVM target compatibility of related compile tasks

In the build module, you may have related compile tasks, for example:

- compileKotlin and compileJava
- compileTestKotlin and compileTestJava

> main and test source set compile tasks are not related.

For related tasks like these, the Kotlin Gradle plugin checks for JVM target compatibility. Different values of the jvmTarget attribute in the kotlin extension or task and targetCompatibility in the java extension or task cause JVM target incompatibility. For example: the compileKotlin task has jvmTarget=1.8, and the compileJava task has (or inherits) targetCompatibility=15.

Configure the behavior of this check by setting the kotlin.jvm.target.validation.mode property in the build.gradle(.kts) file to:

- error – the plugin fails the build; the default value for projects on Gradle 8.0+.
- warning – the plugin prints a warning message; the default value for projects on Gradle less than 8.0.
- ignore – the plugin skips the check and doesn't produce any messages.

To avoid JVM target incompatibility, configure a toolchain or align JVM versions manually.

What can go wrong if not checking targets compatibility

There are two ways of manually setting JVM targets for Kotlin and Java source sets:

- The implicit way via setting up a Java toolchain.
- The explicit way via setting the jvmTarget attribute in the kotlin extension or task and targetCompatibility in the java extension or task.

JVM target incompatibility occurs if you:

- Explicitly set different values of jvmTarget and targetCompatibility.
- Have a default configuration, and your JDK is not equal to 1.8.

Let's consider a default configuration of JVM targets when you have only the Kotlin JVM plugin in your build script and no additional settings for JVM targets:

Kotlin

```kotlin
plugins {
    kotlin("jvm") version "1.9.0"
}
```

Groovy

```groovy
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.9.0"
}
```

When there is no explicit information about the jvmTarget value in the build script, its default value is null, and the compiler translates it to the default value 1.8. The targetCompatibility equals the current Gradle's JDK version, which is equal to your JDK version (unless you use a Java toolchain approach). Assuming that your JDK version is 17, your published library artifact will declare itself compatible with JDK 17+: org.gradle.jvm.version=17, which is wrong. In this case, you have to use Java 17 in your main project to add this library, even though the bytecode's version is 1.8. Configure a toolchain to solve this issue.

Gradle Java toolchains support

> A warning for Android users. To use Gradle toolchain support, use the Android Gradle plugin (AGP) version 8.1.0-alpha09 or higher.
>
> Gradle Java toolchain support is available only from AGP 7.4.0. Nevertheless, because of this issue, AGP did not set targetCompatibility to be equal to the toolchain's JDK until the version 8.1.0-alpha09. If you use versions less than 8.1.0-alpha09, you need to configure targetCompatibility manually via compileOptions. Replace the placeholder <MAJOR_JDK_VERSION> with the JDK version you would like to use:
>
> ```
> android {
>     compileOptions {
>         sourceCompatibility = <MAJOR_JDK_VERSION>
>         targetCompatibility = <MAJOR_JDK_VERSION>
>     }
> }
> ```

Gradle 6.7 introduced Java toolchains support. Using this feature, you can:

- Use a JDK and a JRE that are different from the ones in Gradle to run compilations, tests, and executables.
- Compile and test code with a not-yet-released language version.

With toolchains support, Gradle can autodetect local JDKs and install missing JDKs that Gradle requires for the build. Now Gradle itself can run on any JDK and still reuse the remote build cache feature for tasks that depend on a major JDK version.

The Kotlin Gradle plugin supports Java toolchains for Kotlin/JVM compilation tasks. JS and Native tasks don't use toolchains. The Kotlin compiler always runs on the JDK the Gradle daemon is running on. A Java toolchain:

- Sets the -jdk-home option available for JVM targets.
- Sets the compilerOptions.jvmTarget to the toolchain's JDK version if the user doesn't set the jvmTarget option explicitly. If the user doesn't configure the

toolchain, the jvmTarget field uses the default value. Learn more about JVM target compatibility.

- Sets the toolchain to be used by any Java compile, test and javadoc tasks.
- Affects which JDK kapt workers are running on.

Use the following code to set a toolchain. Replace the placeholder <MAJOR_JDK_VERSION> with the JDK version you would like to use:

Kotlin

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
    // Or shorter:
    jvmToolchain(<MAJOR_JDK_VERSION>)
    // For example:
    jvmToolchain(17)
}
```

Groovy

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
    // Or shorter:
    jvmToolchain(<MAJOR_JDK_VERSION>)
    // For example:
    jvmToolchain(17)
}
```

Note that setting a toolchain via the kotlin extension updates the toolchain for Java compile tasks as well.

You can set a toolchain via the java extension, and Kotlin compilation tasks will use it:

Kotlin

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
}
```

Groovy

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
}
```

If you use Gradle 8.0.2 or higher, you also need to add a toolchain resolver plugin. This type of plugin manages which repositories to download a toolchain from. As an example, add to your settings.gradle(.kts) the following plugin:

Kotlin

```
plugins {
  id("org.gradle.toolchains.foojay-resolver-convention") version("0.5.0")
}
```

Groovy

```
plugins {
  id 'org.gradle.toolchains.foojay-resolver-convention' version '0.5.0'
```

```
}
```

Check that the version of foojay-resolver-convention corresponds to your Gradle version on the Gradle site.

> To understand which toolchain Gradle uses, run your Gradle build with the log level --info and find a string in the output starting with [KOTLIN] Kotlin compilation 'jdkHome' argument:. The part after the colon will be the JDK version from the toolchain.

To set any JDK (even local) for a specific task, use the Task DSL.

Learn more about Gradle JVM toolchain support in the Kotlin plugin.

Set JDK version with the Task DSL

The Task DSL allows setting any JDK version for any task implementing the UsesKotlinJavaToolchain interface. At the moment, these tasks are KotlinCompile and KaptTask. If you want Gradle to search for the major JDK version, replace the <MAJOR_JDK_VERSION> placeholder in your build script:

Kotlin

```kotlin
val service = project.extensions.getByType<JavaToolchainService>()
val customLauncher = service.launcherFor {
    languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
}
project.tasks.withType<UsesKotlinJavaToolchain>().configureEach {
    kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

Groovy

```groovy
JavaToolchainService service = project.getExtensions().getByType(JavaToolchainService.class)
Provider<JavaLauncher> customLauncher = service.launcherFor {
    it.languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
}
tasks.withType(UsesKotlinJavaToolchain::class).configureEach { task ->
    task.kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

Or you can specify the path to your local JDK and replace the placeholder <LOCAL_JDK_VERSION> with this JDK version:

```kotlin
tasks.withType<UsesKotlinJavaToolchain>().configureEach {
    kotlinJavaToolchain.jdk.use(
        "/path/to/local/jdk", // Put a path to your JDK
        JavaVersion.<LOCAL_JDK_VERSION> // For example, JavaVersion.17
    )
}
```

Associate compiler tasks

You can associate compilations by setting up such a relationship between them that one compilation uses the compiled outputs of the other. Associating compilations establishes internal visibility between them.

The Kotlin compiler associates some compilations by default, such as the test and main compilations of each target. If you need to express that one of your custom compilations is connected to another, create your own associated compilation.

To make the IDE support associated compilations for inferring visibility between source sets, add the following code to your build.gradle(.kts):

Kotlin

```kotlin
val integrationTestCompilation = kotlin.target.compilations.create("integrationTest") {
    associateWith(kotlin.target.compilations.getByName("main"))
}
```

Groovy

```groovy
integrationTestCompilation {
    kotlin.target.compilations.create("integrationTest") {
        associateWith(kotlin.target.compilations.getByName("main"))
    }
}
```

Here, the integrationTest compilation is associated with the main compilation that gives access to internal objects from functional tests.

Configure with Java Modules (JPMS) enabled

To make the Kotlin Gradle plugin work with Java Modules, add the following lines to your build script and replace YOUR_MODULE_NAME with a reference to your JPMS module, for example, org.company.module:

Kotlin

```kotlin
// Add the following three lines if you use a Gradle version less than 7.0
java {
    modularity.inferModulePath.set(true)
}

tasks.named("compileJava", JavaCompile::class.java) {
    options.compilerArgumentProviders.add(CommandLineArgumentProvider {
        // Provide compiled Kotlin classes to javac – needed for Java/Kotlin mixed sources to work
        listOf("--patch-module", "YOUR_MODULE_NAME=${sourceSets["main"].output.asPath}")
    })
}
```

Groovy

```groovy
// Add the following three lines if you use a Gradle version less than 7.0
java {
    modularity.inferModulePath = true
}

tasks.named("compileJava", JavaCompile.class) {
    options.compilerArgumentProviders.add(new CommandLineArgumentProvider() {
        @Override
        Iterable<String> asArguments() {
            // Provide compiled Kotlin classes to javac – needed for Java/Kotlin mixed sources to work
            return ["--patch-module", "YOUR_MODULE_NAME=${sourceSets["main"].output.asPath}"]
        }
    })
}
```

Put module-info.java into the src/main/java directory as usual.

For a module, a package name in Kotlin files should be equal to the package name from module-info.java to avoid a "package is empty or does not exist" build failure.

Learn more about:

- Building modules for the Java Module System
- Building applications using the Java Module System
- What "module" means in Kotlin

Other details

Learn more about Kotlin/JVM.

Lazy Kotlin/JVM task creation

Starting from Kotlin 1.8.20, the Kotlin Gradle plugin registers all tasks and doesn't configure them on a dry run.

Non-default location of compile tasks' destinationDirectory

If you override the Kotlin/JVM KotlinJvmCompile/KotlinCompile task's destinationDirectory location, update your build script. You need to explicitly add sourceSets.main.kotlin.classesDirectories to sourceSets.main.outputs in your JAR file:

```
tasks.jar(type: Jar) {
     from sourceSets.main.outputs
     from sourceSets.main.kotlin.classesDirectories
}
```

Targeting multiple platforms

Projects targeting multiple platforms, called multiplatform projects, require the kotlin-multiplatform plugin. Learn more about the plugin.

> The kotlin-multiplatform plugin works with Gradle 6.8.3 or later.

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}
```

Learn more about Kotlin Multiplatform for different platforms and Kotlin Multiplatform for iOS and Android.

Targeting Android

It's recommended to use Android Studio for creating Android applications. Learn how to use the Android Gradle plugin.

Targeting JavaScript

When targeting JavaScript, use the kotlin-multiplatform plugin as well. Learn more about setting up a Kotlin/JS project

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.0"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.0'
}
```

Kotlin and Java sources for JavaScript

This plugin only works for Kotlin files, so it is recommended that you keep Kotlin and Java files separate (if the project contains Java files). If you don't store them separately, specify the source folder in the sourceSets block:

Kotlin

```kotlin
kotlin {
    sourceSets["main"].apply {
        kotlin.srcDir("src/main/myKotlin")
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        main.kotlin.srcDirs += 'src/main/myKotlin'
    }
}
```

Triggering configuration actions with the KotlinBasePlugin interface

To trigger some configuration action whenever any Kotlin Gradle plugin (JVM, JS, Multiplatform, Native, and others) is applied, use the KotlinBasePlugin interface that all Kotlin plugins inherit from:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin

// ...

project.plugins.withType<KotlinBasePlugin>() {
    // Configure your action here
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin

// ...

project.plugins.withType(KotlinBasePlugin.class) {
    // Configure your action here
}
```

Configure dependencies

To add a dependency on a library, set the dependency of the required type (for example, implementation) in the dependencies block of the source sets DSL.

Kotlin

```kotlin
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("com.example:my-library:1.0")
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'com.example:my-library:1.0'
```

```
            }
        }
    }
}
```

Alternatively, you can set dependencies at top level.

Dependency types

Choose the dependency type based on your requirements.

Type	Description	When to use
api	Used both during compilation and at runtime and is exported to library consumers.	If any type from a dependency is used in the public API of the current module, use an api dependency.
implementation	Used during compilation and at runtime for the current module, but is not exposed for compilation of other modules depending on the one with the `implementation` dependency.	Use for dependencies needed for the internal logic of a module. If a module is an endpoint application which is not published, use implementation dependencies instead of api dependencies.
compileOnly	Used for compilation of the current module and is not available at runtime nor during compilation of other modules.	Use for APIs which have a third-party implementation available at runtime.
runtimeOnly	Available at runtime but is not visible during compilation of any module.	

Dependency on the standard library

A dependency on the standard library (stdlib) is added automatically to each source set. The version of the standard library used is the same as the version of the Kotlin Gradle plugin.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin selects the appropriate JVM standard library depending on the compilerOptions.jvmTarget compiler option of your Gradle build script.

If you declare a standard library dependency explicitly (for example, if you need a different version), the Kotlin Gradle plugin won't override it or add a second standard library.

If you do not need a standard library at all, you can add the opt-out option to the gradle.properties:

```
kotlin.stdlib.default.dependency=false
```

Versions alignment of transitive dependencies

If you explicitly write the Kotlin version 1.8.0 or higher in your dependencies, for example: implementation("org.jetbrains.kotlin:kotlin-stdlib:1.8.0"), then the Kotlin Gradle Plugin uses this Kotlin version for transitive kotlin-stdlib-jdk7 and kotlin-stdlib-jdk8 dependencies. This is for avoiding class duplication from different stdlib versions. Learn more about merging kotlin-stdlib-jdk7 and kotlin-stdlib-jdk8 into kotlin-stdlib. You can disable this behavior with the kotlin.stdlib.jdk.variants.version.alignment Gradle property:

```
kotlin.stdlib.jdk.variants.version.alignment=false
```

Other ways to align versions

- In case you have issues with versions alignment, align all versions via the Kotlin BOM. Declare a platform dependency on kotlin-bom in your build script:

Kotlin

```
implementation(platform("org.jetbrains.kotlin:kotlin-bom:1.9.0"))
```

Groovy

```
implementation platform('org.jetbrains.kotlin:kotlin-bom:1.9.0')
```

- If you don't have a standard library explicitly: kotlin.stdlib.default.dependency=false in your gradle.properties, but one of your dependencies transitively brings some old Kotlin stdlib version, for example, kotlin-stdlib-jdk7:1.7.20 and another dependency transitively brings kotlin-stdlib:1.8+ – in this case, you can require 1.9.0 versions of these transitive libraries:

Kotlin

```
dependencies {
    constraints {
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk7") {
            version {
                require("1.9.0")
            }
        }
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk8") {
            version {
                require("1.9.0")
            }
        }
    }
}
```

Groovy

```
dependencies {
    constraints {
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk7") {
            version {
                require("1.9.0")
            }
        }
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk8") {
            version {
                require("1.9.0")
            }
        }
    }
}
```

- If you have a Kotlin version equal to 1.9.0: implementation("org.jetbrains.kotlin:kotlin-stdlib:1.9.0") and an old version (less than 1.8.0) of a Kotlin Gradle plugin – update the Kotlin Gradle plugin:

Kotlin

```
plugins {
    // replace `<...>` with the plugin name
    kotlin("<...>") version "1.9.0"
}
```

Groovy

```
plugins {
    // replace `<...>` with the plugin name
    id "org.jetbrains.kotlin.<...>" version "1.9.0"
}
```

* If you have an explicit old version (less than 1.8.0) of kotlin-stdlib-jdk7/kotlin-stdlib-jdk8, for example, implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk7:SOME_OLD_KOTLIN_VERSION"), and a dependency that transitively brings kotlin-stdlib:1.8+, replace your kotlin-stdlib-jdk<7/8>:SOME_OLD_KOTLIN_VERSION with kotlin-stdlib-jdk*:1.9.0 or exclude a transitive kotlin-stdlib:1.8+ from the library that brings it:

  Kotlin

  ```
  dependencies {
      implementation("com.example:lib:1.0") {
          exclude(group = "org.jetbrains.kotlin", module = "kotlin-stdlib")
      }
  }
  ```

  Groovy

  ```
  dependencies {
      implementation("com.example:lib:1.0") {
          exclude group: "org.jetbrains.kotlin", module: "kotlin-stdlib"
      }
  }
  ```

Set dependencies on test libraries

The kotlin.test API is available for testing Kotlin projects on all supported platforms. Add the dependency kotlin-test to the commonTest source set, so that the Gradle plugin can infer the corresponding test dependencies for each test source set:

* kotlin-test-common and kotlin-test-annotations-common for common source sets
* kotlin-test-junit for JVM source sets
* kotlin-test-js for Kotlin/JS source sets

Kotlin/Native targets do not require additional test dependencies, and the kotlin.test API implementations are built-in.

Kotlin

```
kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // This brings all the platform dependencies automatically
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings all the platform dependencies automatically
            }
        }
    }
}
```

> You can use shorthand for a dependency on a Kotlin module, for example, kotlin("test") for "org.jetbrains.kotlin:kotlin-test".

You can use the kotlin-test dependency in any shared or platform-specific source set as well.

For Kotlin/JVM, Gradle uses JUnit 4 by default. Therefore, the kotlin("test") dependency resolves to the variant for JUnit 4, namely kotlin-test-junit.

You can choose JUnit 5 or TestNG by calling useJUnitPlatform() or useTestNG() in the test task of your build script. The following example is for a Kotlin

Multiplatform project:

Kotlin

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
    }
}
```

Groovy

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test")
            }
        }
    }
}
```

The following example is for a JVM project:

Kotlin

```
dependencies {
    testImplementation(kotlin("test"))
}

tasks {
    test {
        useTestNG()
    }
}
```

Groovy

```
dependencies {
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}

test {
    useTestNG()
}
```

Learn how to test code using JUnit on the JVM.

If you need to use a different JVM test framework, disable automatic testing framework selection by adding the line kotlin.test.infer.jvm.variant=false to the project's gradle.properties file. After doing this, add the framework as a Gradle dependency.

If you have used a variant of kotlin("test") in your build script explicitly and your project build stopped working with a compatibility conflict, see this issue in the Compatibility Guide.

Set a dependency on a kotlinx library

If you use a kotlinx library and need a platform-specific dependency, you can use platform-specific variants of libraries with suffixes such as -jvm or -js, for example, kotlinx-coroutines-core-jvm. You can also use the library's base artifact name instead – kotlinx-coroutines-core.

Kotlin

```kotlin
kotlin {
    sourceSets {
        val jvmMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.7.1")
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        jvmMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.7.1'
            }
        }
    }
}
```

If you use a multiplatform library and need to depend on the shared code, set the dependency only once, in the shared source set. Use the library's base artifact name, such as kotlinx-coroutines-core or ktor-client-core.

Kotlin

```kotlin
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1'
            }
        }
    }
}
```

Set dependencies at top level

Alternatively, you can specify the dependencies at top level, using the following pattern for the configuration names: <sourceSetName><DependencyType>. This can be helpful for some Gradle built-in dependencies, like gradleApi(), localGroovy(), or gradleTestKit(), which are not available in the source sets' dependency DSL.

Kotlin

```kotlin
dependencies {
    "commonMainImplementation"("com.example:my-library:1.0")
}
```

Groovy

```groovy
dependencies {
    commonMainImplementation 'com.example:my-library:1.0'
}
```

What's next?

Learn more about:

- Compiler options and how to pass them.
- Incremental compilation, caches support, build reports, and the Kotlin daemon.
- Gradle basics and specifics.
- Support for Gradle plugin variants.

Compiler options in the Kotlin Gradle plugin

Each release of Kotlin includes compilers for the supported targets: JVM, JavaScript, and native binaries for supported platforms.

These compilers are used by:

- The IDE, when you click the Compile or Run button for your Kotlin project.
- Gradle, when you call gradle build in a console or in the IDE.
- Maven, when you call mvn compile or mvn test-compile in a console or in the IDE.

You can also run Kotlin compilers manually from the command line as described in the Working with command-line compiler tutorial.

How to define options

Kotlin compilers have a number of options for tailoring the compiling process.

Using a build script, you can specify additional compilation options. Use the compilerOptions property of a Kotlin compilation task for it. For example:

Kotlin

```kotlin
tasks.named("compileKotlin", org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask::class.java) {
    compilerOptions {
        freeCompilerArgs.add("-Xexport-kdoc")
    }
}
```

Groovy

```groovy
tasks.named('compileKotlin', org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class) {
    compilerOptions {
        freeCompilerArgs.add("-Xexport-kdoc")
    }
}
```

When targeting the JVM, the tasks are called compileKotlin for production code and compileTestKotlin for test code. The tasks for custom source sets are named according to their compile<Name>Kotlin patterns.

The names of the tasks in Android Projects contain build variant names and follow the compile<BuildVariant>Kotlin pattern, for example, compileDebugKotlin or compileReleaseUnitTestKotlin.

When targeting JavaScript, the tasks are called compileKotlinJs for production code and compileTestKotlinJs for test code, and compile<Name>KotlinJs for custom source sets.

To configure a single task, use its name. Examples:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

val compileKotlin: KotlinCompilationTask<*> by tasks

compileKotlin.compilerOptions.suppressWarnings.set(true)
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        suppressWarnings.set(true)
    }
}
```

Note that with the Gradle Kotlin DSL, you should get the task from the project's tasks first.

Use the Kotlin2JsCompile and KotlinCompileCommon types for JS and common targets, respectively.

It is also possible to configure all of the Kotlin compilation tasks in the project:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
    compilerOptions { /*...*/ }
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions { /*...*/ }
}
```

Here is a complete list of options for Gradle tasks:

Attributes specific to JVM

Name	Description	Possible values	Default value
javaParameters	Generate metadata for Java 1.8 reflection on method parameters		false
jvmTarget	Target version of the generated JVM bytecode	"1.8", "9", "10", ..., "20". Also, see Types for compiler options	"1.8"

Name	Description	Possible values	Default value
noJdk	Don't automatically include the Java runtime into the classpath		false

Attributes common to JVM, JS, and JS DCE

Name	Description	Possible values	Default value
allWarningsAsErrors	Report an error if there are any warnings		false
suppressWarnings	Don't generate warnings		false
verbose	Enable verbose logging output. Works only when the Gradle debug log level enabled		false
freeCompilerArgs	A list of additional compiler arguments. You can use experimental -X arguments here too. See an example		[]

We are going to deprecate the attribute freeCompilerArgs in future releases. If you miss some option in the Kotlin Gradle DSL, please, file an issue.

Example of additional arguments usage via freeCompilerArgs

Use the attribute freeCompilerArgs to supply additional (including experimental) compiler arguments. You can add a single argument to this attribute or a list of arguments:

Kotlin

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

val compileKotlin: KotlinCompilationTask<*> by tasks

// Single experimental argument
compileKotlin.compilerOptions.freeCompilerArgs.add("-Xexport-kdoc")
// Single additional argument, can be a key-value pair
compileKotlin.compilerOptions.freeCompilerArgs.add("-opt-in=org.mylibrary.OptInAnnotation")
// List of arguments
compileKotlin.compilerOptions.freeCompilerArgs.addAll(listOf("-Xno-param-assertions", "-Xno-receiver-assertions", "-Xno-call-
assertions"))
```

Groovy

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        // Single experimental argument
        freeCompilerArgs.add("-Xexport-kdoc")
        // Single additional argument, can be a key-value pair
        freeCompilerArgs.add("-opt-in=org.mylibrary.OptInAnnotation")
        // List of arguments
        freeCompilerArgs.addAll(["-Xno-param-assertions", "-Xno-receiver-assertions", "-Xno-call-assertions"])
    }
}
```

Attributes common to JVM and JS

Name	Description	Possible values	Default value
apiVersion	Restrict the use of declarations to those from the specified version of bundled libraries	"1.3" (DEPRECATED), "1.4" (DEPRECATED), "1.5", "1.6", "1.7", "1.8", "1.9" (EXPERIMENTAL)	
languageVersion	Provide source compatibility with the specified version of Kotlin	"1.3" (DEPRECATED), "1.4" (DEPRECATED), "1.5", "1.6", "1.7", "1.8", "1.9" (EXPERIMENTAL)	

Example of setting a languageVersion

To set a language version, use the following syntax:

Kotlin

```kotlin
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```groovy
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
}
```

Also, see Types for compiler options.

Attributes specific to JS

Name	Description	Possible values	Default value
friendModulesDisabled	Disable internal declaration export		false
main	Define whether the main function should be called upon execution	"call", "noCall". Also, see Types for compiler options	"call"
metaInfo	Generate .meta.js and .kjsm files with metadata. Use to create a library		true

Name	Description	Possible values	Default value
moduleKind	The kind of JS module generated by the compiler	"umd", "commonjs", "amd", "plain", "es". Also, see Types for compiler options	"umd"
outputFile	Destination *.js file for the compilation result		"<buildDir>/js/packages/<project.name>/kotlin/<project.name>.js"
sourceMap	Generate source map		true
sourceMapEmbedSources	Embed source files into the source map	"never", "always", "inlining". Also, see Types for compiler options	
sourceMapNamesPolicy	Add variable and function names that you declared in Kotlin code into the source map. For more information on the behavior, see our compiler reference.	"simple-names", "fully-qualified-names", "no". Also, see Types for compiler options	"simple-names"
sourceMapPrefix	Add the specified prefix to paths in the source map		
target	Generate JS files for specific ECMA version	"v5"	"v5"
typedArrays	Translate primitive arrays to JS typed arrays		true

Types for compiler options

Some of the compilerOptions use the new types instead of the String type:

Option	Type	Example
jvmTarget	JvmTarget	compilerOptions.jvmTarget.set(JvmTarget.JVM_11)
apiVersion and languageVersion	KotlinVersion	compilerOptions.languageVersion.set(KotlinVersion.KOTLIN_1_9)
main	JsMainFunctionExecutionMode	compilerOptions.main.set(JsMainFunctionExecutionMode.NO_CALL)
moduleKind	JsModuleKind	compilerOptions.moduleKind.set(JsModuleKind.MODULE_ES)

Option	Type	Example
sourceMapEmbedSources	JsSourceMapEmbedMode	compilerOptions.sourceMapEmbedSources.set(JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CO
sourceMapNamesPolicy	JsSourceMapNamesPolicy	compilerOptions.sourceMapNamesPolicy.set(JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY

What's next?

Learn more about:

- Incremental compilation, caches support, build reports, and the Kotlin daemon.
- Gradle basics and specifics.
- Support for Gradle plugin variants.

Compilation and caches in the Kotlin Gradle plugin

On this page, you can learn about the following topics:

- Incremental compilation
- Gradle build cache support
- Gradle configuration cache support
- The Kotlin daemon and how to use it with Gradle
- Defining Kotlin compiler execution strategy
- Kotlin compiler fallback strategy
- Build reports

Incremental compilation

The Kotlin Gradle plugin supports incremental compilation. Incremental compilation tracks changes to source files between builds so that only the files affected by these changes are compiled.

Incremental compilation is supported for Kotlin/JVM and Kotlin/JS projects, and is enabled by default.

There are several ways to disable incremental compilation:

- Set kotlin.incremental=false for Kotlin/JVM.
- Set kotlin.incremental.js=false for Kotlin/JS projects.
- Use -Pkotlin.incremental=false or -Pkotlin.incremental.js=false as a command line parameter.

  The parameter should be added to each subsequent build.

Note: Any build with incremental compilation disabled invalidates incremental caches. The first build is never incremental.

> Sometimes problems with incremental compilation become visible several rounds after the failure occurs. Use build reports to track the history of changes and compilations. This can help you to provide reproducible bug reports.

A new approach to incremental compilation

The new approach to incremental compilation is available since Kotlin 1.7.0 for the JVM backend in the Gradle build system only. Starting from Kotlin 1.8.20, this is enabled by default. This approach supports changes made inside dependent non-Kotlin modules, includes an improved compilation avoidance, and is compatible with the Gradle build cache.

All of these enhancements decrease the number of non-incremental builds, making the overall compilation time faster. You will receive the most benefit if you use the build cache, or, frequently make changes in non-Kotlin Gradle modules.

To opt out from this new approach, set the following option in your gradle.properties:

```
kotlin.incremental.useClasspathSnapshot=false
```

We would appreciate your feedback on this feature in YouTrack.

Learn how the new approach to incremental compilation is implemented under the hood in this blog post.

Precise backup of compilation tasks' outputs

> Precise backup of compilation tasks' outputs is Experimental. We would appreciate your feedback on it in YouTrack.

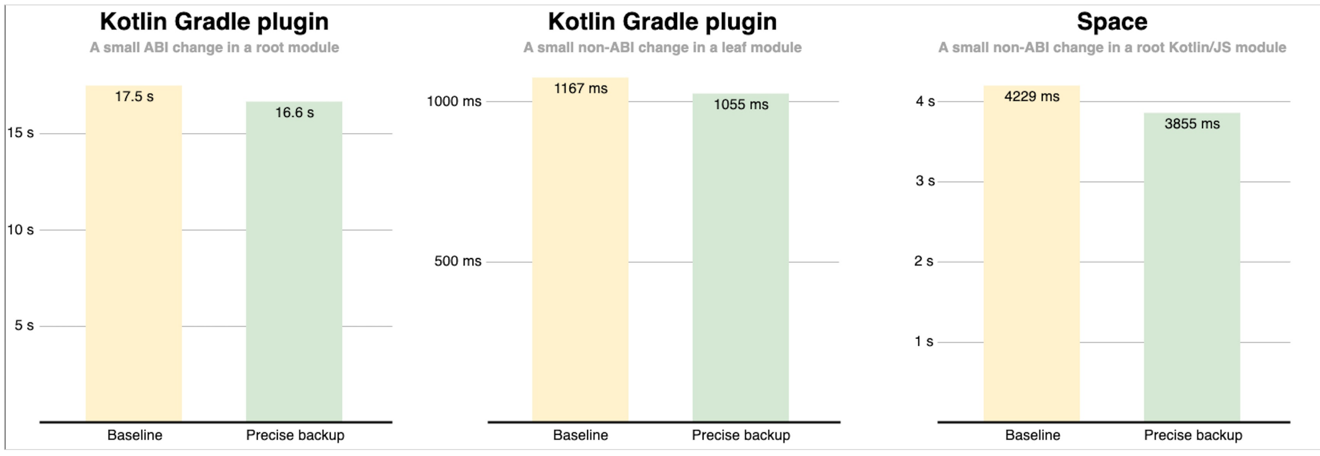
Starting with Kotlin 1.8.20, you can enable precise backup, whereby only those classes that Kotlin recompiles in the incremental compilation are backed up. Both full and precise backups help to run builds incrementally again after compilation errors. A precise backup takes less build time compared to a full backup. A full backup may take noticeably more build time in large projects or if many tasks are creating backups, especially if a project is located on a slow HDD.

Enable this optimization by adding the kotlin.compiler.preciseCompilationResultsBackup Gradle property to the gradle.properties file:

```
kotlin.compiler.preciseCompilationResultsBackup=true
```

Example of using precise backup at JetBrains

In the following charts, you can see examples of using precise backup compared to full backup:



Comparison of full and precise backups

The first and second charts show how using precise backup in a Kotlin project affects building the Kotlin Gradle plugin:

1. After making a small ABI change: adding a new public method to a module that lots of modules depend on.
2. After making a small non-ABI change: adding a private function to a module that no other modules depend on.

The third chart shows how precise backup in the Space project affects building a web frontend after a small non-ABI change: adding a private function to a Kotlin/JS module that lots of modules depend on.

These measurements were performed on a computer with an Apple M1 Max CPU; different computers will yield slightly different results. The factors affecting performance include but are not limited to:

- How warm the Kotlin daemon and the Gradle daemon are.
- How fast or slow the disk is.
- The CPU model and how busy it is.
- Which modules are affected by the changes and how big these modules are.
- Whether the changes are ABI or non-ABI.

Evaluating optimizations with build reports

To estimate the impact of the optimization on your computer for your project and your scenarios, you can use Kotlin build reports. Enable reports in text file format by adding the following property to your gradle.properties file:

kotlin.build.report.output=file

Here is an example of a relevant part of the report before enabling precise backup:

Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.59 s <...> Time metrics: Total Gradle task time: 0.59 s Task action before worker execution: 0.24 s Backup output: 0.22 s // Pay attention to this number <...>

And here is an example of a relevant part of the report after enabling precise backup:

Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.46 s <...> Time metrics: Total Gradle task time: 0.46 s Task action before worker execution: 0.07 s Backup output: 0.05 s // The time has reduced Run compilation in Gradle worker: 0.32 s Clear jar cache: 0.00 s Precise backup output: 0.00 s // Related to precise backup Cleaning up the backup stash: 0.00 s // Related to precise backup <...>

Gradle build cache support

The Kotlin plugin uses the Gradle build cache, which stores the build outputs for reuse in future builds.

To disable caching for all Kotlin tasks, set the system property kotlin.caching.enabled to false (run the build with the argument -Dkotlin.caching.enabled=false).

Gradle configuration cache support

> Gradle configuration cache support has some constraints:
>
> - The configuration cache is available in Gradle 6.5 and later as an experimental feature.
>   You can check the Gradle releases page to see whether it has been promoted to stable.
> - The feature is supported only by the following Gradle plugins:
>   - org.jetbrains.kotlin.jvm
>   - org.jetbrains.kotlin.js
>   - org.jetbrains.kotlin.android

The Kotlin plugin uses the Gradle configuration cache, which speeds up the build process by reusing the results of the configuration phase.

See the Gradle documentation to learn how to enable the configuration cache. After you enable this feature, the Kotlin Gradle plugin automatically starts using it.

The Kotlin daemon and how to use it with Gradle

The Kotlin daemon:

- Runs with the Gradle daemon to compile the project.
- Runs separately from the Gradle daemon when you compile the project with an IntelliJ IDEA built-in build system.

The Kotlin daemon starts at the Gradle execution stage when one of the Kotlin compile tasks starts to compile sources. The Kotlin daemon stops either with the Gradle daemon or after two idle hours with no Kotlin compilation.

The Kotlin daemon uses the same JDK that the Gradle daemon does.

Setting Kotlin daemon's JVM arguments

Each of the following ways to set arguments overrides the ones that came before it:

- Gradle daemon arguments inheritance
- kotlin.daemon.jvm.options system property
- kotlin.daemon.jvmargs property
- kotlin extension
- Specific task definition

Gradle daemon arguments inheritance

If nothing is specified, the Kotlin daemon inherits arguments from the Gradle daemon. For example, in the gradle.properties file:

```
org.gradle.jvmargs=-Xmx1500m -Xms=500m
```

kotlin.daemon.jvm.options system property

If the Gradle daemon's JVM arguments have the kotlin.daemon.jvm.options system property – use it in the gradle.properties file:

```
org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m,Xms=500m
```

When passing arguments, follow these rules:

- Use the minus sign - only before the arguments Xmx, XX:MaxMetaspaceSize, and XX:ReservedCodeCacheSize.
- Separate arguments with commas (,) without spaces. Arguments that come after a space will be used for the Gradle daemon, not for the Kotlin daemon.

> Gradle ignores these properties if all the following conditions are satisfied:
>
> - Gradle is using JDK 1.9 or higher.
> - The version of Gradle is between 7.0 and 7.1.1 inclusively.
> - Gradle is compiling Kotlin DSL scripts.
> - The Kotlin daemon isn't running.
>
> To overcome this, upgrade Gradle to the version 7.2 (or higher) or use the kotlin.daemon.jvmargs property – see the following section.

kotlin.daemon.jvmargs property

You can add the kotlin.daemon.jvmargs property in the gradle.properties file:

```
kotlin.daemon.jvmargs=-Xmx1500m -Xms=500m
```

kotlin extension

You can specify arguments in the kotlin extension:

Kotlin

```kotlin
kotlin {
    kotlinDaemonJvmArgs = listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC")
}
```

Groovy

```groovy
kotlin {
    kotlinDaemonJvmArgs = ["-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"]
}
```

Specific task definition

You can specify arguments for a specific task:

Kotlin

```kotlin
tasks.withType<CompileUsingKotlinDaemon>().configureEach {
    kotlinDaemonJvmArguments.set(listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"))
}
```

Groovy

```groovy
tasks.withType(CompileUsingKotlinDaemon::class).configureEach { task ->
    task.kotlinDaemonJvmArguments.set(["-Xmx1g", "-Xms512m"])
}
```

> In this case a new Kotlin daemon instance can start on task execution. Learn more about Kotlin daemon's behavior with JVM arguments.

Kotlin daemon's behavior with JVM arguments

When configuring the Kotlin daemon's JVM arguments, note that:

- It is expected to have multiple instances of the Kotlin daemon running at the same time when different subprojects or tasks have different sets of JVM arguments.
- A new Kotlin daemon instance starts only when Gradle runs a related compilation task and existing Kotlin daemons do not have the same set of JVM arguments. Imagine that your project has a lot of subprojects. Most of them require some heap memory for a Kotlin daemon, but one module requires a lot (though it is rarely compiled). In this case, you should provide a different set of JVM arguments for such a module, so a Kotlin daemon with a larger heap size would start only for developers who touch this specific module.

  > If you are already running a Kotlin daemon that has enough heap size to handle the compilation request, even if other requested JVM arguments are different, this daemon will be reused instead of starting a new one.

- If the Xmx argument is not specified, the Kotlin daemon will inherit it from the Gradle daemon.

Defining Kotlin compiler execution strategy

Kotlin compiler execution strategy defines where the Kotlin compiler is executed and if incremental compilation is supported in each case.

There are three compiler execution strategies:

Strategy	Where Kotlin compiler is executed	Incremental compilation	Other characteristics and notes
Daemon	Inside its own daemon process	Yes	The default and fastest strategy. Can be shared between different Gradle daemons and multiple parallel compilations.

Strategy	Where Kotlin compiler is executed	Incremental compilation	Other characteristics and notes
In process	Inside the Gradle daemon process	No	May share the heap with the Gradle daemon. The "In process" execution strategy is slower than the "Daemon" execution strategy. Each worker creates a separate Kotlin compiler classloader for each compilation.
Out of process	In a separate process for each compilation	No	The slowest execution strategy. Similar to the "In process", but additionally creates a separate Java process within a Gradle worker for each compilation.

To define a Kotlin compiler execution strategy, you can use one of the following properties:

- The kotlin.compiler.execution.strategy Gradle property.
- The compilerExecutionStrategy compile task property.

The task property compilerExecutionStrategy takes priority over the Gradle property kotlin.compiler.execution.strategy.

The available values for the kotlin.compiler.execution.strategy property are:

1. daemon (default)
2. in-process
3. out-of-process

Use the Gradle property kotlin.compiler.execution.strategy in gradle.properties:

```
kotlin.compiler.execution.strategy=out-of-process
```

The available values for the compilerExecutionStrategy task property are:

1. org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.DAEMON (default)
2. org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.IN_PROCESS
3. org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.OUT_OF_PROCESS

Use the task property compilerExecutionStrategy in your build scripts:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.CompileUsingKotlinDaemon
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType<CompileUsingKotlinDaemon>().configureEach {
    compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN_PROCESS)
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.CompileUsingKotlinDaemon
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType(CompileUsingKotlinDaemon)
    .configureEach {
        compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN_PROCESS)
    }
```

Kotlin compiler fallback strategy

The Kotlin compiler's fallback strategy is to run a compilation outside a Kotlin daemon if the daemon somehow fails. If the Gradle daemon is on, the compiler uses the "In process" strategy. If the Gradle daemon is off, the compiler uses the "Out of process" strategy.

When this fallback happens, you have the following warning lines in your Gradle's build output:

```
Failed to compile with Kotlin daemon: java.lang.RuntimeException: Could not connect to Kotlin compile daemon
[exception stacktrace]
Using fallback strategy: Compile without Kotlin daemon
Try ./gradlew --stop if this issue persists.
```

However, a silent fallback to another strategy can consume a lot of system resources or lead to non-deterministic builds. Read more about this in this YouTrack issue. To avoid this, there is a Gradle property kotlin.daemon.useFallbackStrategy, whose default value is true. When the value is false, builds fail on problems with the daemon's startup or communication. Declare this property in gradle.properties:

```
kotlin.daemon.useFallbackStrategy=false
```

There is also a useDaemonFallbackStrategy property in Kotlin compile tasks, which takes priority over the Gradle property if you use both.

Kotlin

```
tasks {
    compileKotlin {
        useDaemonFallbackStrategy.set(false)
    }
}
```

Groovy

```
tasks.named("compileKotlin").configure {
    useDaemonFallbackStrategy = false
}
```

If there is insufficient memory to run the compilation, you can see a message about it in the logs.

Build reports

> Build reports are Experimental. They may be dropped or changed at any time. Opt-in is required (see details below). Use them only for evaluation purposes. We appreciate your feedback on them in YouTrack.

Build reports for tracking compiler performance are available starting from Kotlin 1.7.0. Reports contain the durations of different compilation phases and reasons why compilation couldn't be incremental.

Use build reports to investigate performance issues when the compilation time is too long or when it differs for the same project.

Kotlin build reports help examine problems more efficiently than Gradle build scans. Lots of engineers use them to investigate build performance, but the unit of granularity in Gradle scans is a single Gradle task.

There are two common cases that analyzing build reports for long-running compilations can help you resolve:

- The build wasn't incremental. Analyze the reasons and fix underlying problems.
- The build was incremental but took too much time. Try reorganizing source files — split big files, save separate classes in different files, refactor large classes, declare top-level functions in different files, and so on.

Learn how to read build reports and how JetBrains uses build reports.

Enabling build reports

To enable build reports, declare where to save the build report output in gradle.properties:

```
kotlin.build.report.output=file
```

The following values and their combinations are available for the output:

Option	Description
file	Saves build reports in a human-readable format to a local file. By default, it's ${project_folder}/build/reports/kotlin-build/${project_name}-timestamp.txt
single_file	Saves build reports in a format of an object to a specified local file
build_scan	Saves build reports in the custom values section of the build scan. Note that the Gradle Enterprise plugin limits the number of custom values and their length. In big projects, some values could be lost
http	Posts build reports using HTTP(S). The POST method sends metrics in JSON format. You can see the current version of the sent data in the Kotlin repository. You can find samples of HTTP endpoints in this blog post

Here's a list of available options for kotlin.build.report:

```
# Required outputs. Any combination is allowed
kotlin.build.report.output=file,single_file,http,build_scan

# Mandatory if single_file output is used. Where to put reports
# Use instead of the deprecated `kotlin.internal.single.build.metrics.file` property
kotlin.build.report.single_file=some_filename

# Optional. Output directory for file-based reports. Default: build/reports/kotlin-build/
kotlin.build.report.file.output_dir=kotlin-reports

# Optional. Label for marking your build report (for example, debug parameters)
kotlin.build.report.label=some_label
```

Options, applicable only to HTTP:

```
# Mandatory. Where to post HTTP(S)-based reports
kotlin.build.report.http.url=http://127.0.0.1:8080

# Optional. User and password if the HTTP endpoint requires authentication
kotlin.build.report.http.user=someUser
kotlin.build.report.http.password=somePassword

# Optional. Add a Git branch name of a build to a build report
kotlin.build.report.http.include_git_branch.name=true|false

# Optional. Add compiler arguments to a build report
# If a project contains many modules, its compiler arguments in the report can be very heavy and not that helpful
kotlin.build.report.include_compiler_arguments=true|false
```

Limit of custom values

To collect build scans' statistics, Kotlin build reports use Gradle's custom values. Both you and different Gradle plugins can write data to custom values. The number of custom values has a limit. See the current maximum custom value count in the Build scan plugin docs.

If you have a big project, a number of such custom values may be quite big. If this number exceeds the limit, you can see the following message in the logs:

```
Maximum number of custom values (1,000) exceeded
```

To reduce the number of custom values the Kotlin plugin produces, you can use the following property in gradle.properties:

```
kotlin.build.report.build_scan.custom_values_limit=500
```

Switching off collecting project and system properties
HTTP build statistic logs can contain some project and system properties. These properties can change builds' behavior, so it's useful to log them in build statistics. These properties can store sensitive data, for example, passwords or a project's full path.

You can disable collection of these statistics by adding the kotlin.build.report.http.verbose_environment property to your gradle.properties.

> JetBrains doesn't collect these statistics. You choose a place where to store your reports.

What's next?

Learn more about:

- Gradle basics and specifics.
- Support for Gradle plugin variants.

Support for Gradle plugin variants

Gradle 7.0 introduced a new feature for Gradle plugin authors — plugins with variants. This feature makes it easier to add support for latest Gradle features while maintaining compatibility with older Gradle versions. Learn more about variant selection in Gradle.

With Gradle plugin variants, the Kotlin team can ship different Kotlin Gradle plugin (KGP) variants for different Gradle versions. The goal is to support the base Kotlin compilation in the main variant, which corresponds to the oldest supported versions of Gradle. Each variant will have implementations for Gradle features from a corresponding release. The latest variant will support the latest Gradle feature set. With this approach, it is possible to extend support for older Gradle versions with limited functionality.

Currently, there are the following variants of the Kotlin Gradle plugin:

Variant's name	Corresponding Gradle versions
main	6.8.3–6.9.3
gradle70	7.0
gradle71	7.1-7.4
gradle75	7.5
gradle76	7.6 and higher

In future Kotlin releases, more variants will probably be added.

To check which variant your build uses, enable the --info log level and find a string in the output starting with Using Kotlin Gradle plugin, for example, Using Kotlin Gradle plugin main variant.

Troubleshooting

Here are workarounds for some known issues with variant selection in Gradle:

- ResolutionStrategy in pluginManagement is not working for plugins with multivariants
- Plugin variants are ignored when a plugin is added as the buildSrc common dependency

Gradle can't select a KGP variant in a custom configuration

This is an expected situation that Gradle can't select a KGP variant in a custom configuration. If you use a custom Gradle configuration:

Kotlin

```
configurations.register("customConfiguration") {
    // ...
}
```

Groovy

```
configurations.register("customConfiguration") {
    // ...
}
```

And want to add a dependency on the Kotlin Gradle plugin, for example:

Kotlin

```
dependencies {
    customConfiguration("org.jetbrains.kotlin:kotlin-gradle-plugin:1.9.0")
}
```

Groovy

```
dependencies {
    customConfiguration 'org.jetbrains.kotlin:kotlin-gradle-plugin:1.9.0'
}
```

You need to add the following attributes to your customConfiguration:

Kotlin

```
configurations {
    customConfiguration {
        attributes {
            attribute(
                Usage.USAGE_ATTRIBUTE,
                project.objects.named(Usage.class, Usage.JAVA_RUNTIME)
            )
            attribute(
                Category.CATEGORY_ATTRIBUTE,
                project.objects.named(Category.class, Category.LIBRARY)
            )
            // If you want to depend on a specific KGP variant:
            attribute(
                GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                project.objects.named("7.0")
            )
        }
    }
}
```

Groovy

```
configurations {
    customConfiguration {
        attributes {
            attribute(
                Usage.USAGE_ATTRIBUTE,
                project.objects.named(Usage, Usage.JAVA_RUNTIME)
            )
            attribute(
                Category.CATEGORY_ATTRIBUTE,
                project.objects.named(Category, Category.LIBRARY)
            )
            // If you want to depend on a specific KGP variant:
            attribute(
                GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                project.objects.named('7.0')
            )
        }
    }
}
```

Otherwise, you will receive an error similar to this:

```
 > Could not resolve all files for configuration ':customConfiguration'.
      > Could not resolve org.jetbrains.kotlin:kotlin-gradle-plugin:1.7.0.
        Required by:
            project :
         > Cannot choose between the following variants of org.jetbrains.kotlin:kotlin-gradle-plugin:1.7.0:
             - gradle70RuntimeElements
             - runtimeElements
           All of them match the consumer attributes:
             - Variant 'gradle70RuntimeElements' capability org.jetbrains.kotlin:kotlin-gradle-plugin:1.7.0:
                 - Unmatched attributes:
```

What's next?

Learn more about Gradle basics and specifics.

Maven

Plugin and versions

The kotlin-maven-plugin compiles Kotlin sources and modules. Currently, only Maven v3 is supported.

Define the version of Kotlin you want to use via a kotlin.version property:

```
<properties>
    <kotlin.version>1.9.0</kotlin.version>
</properties>
```

Dependencies

Kotlin has an extensive standard library that can be used in your applications. To use the standard library in your project, add the following dependency in the pom file:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

> If you're targeting JDK 7 or 8 with Kotlin versions older than:
>
> - 1.8, use kotlin-stdlib-jdk7 or kotlin-stdlib-jdk8, respectively.
> - 1.2, use kotlin-stdlib-jre7 or kotlin-stdlib-jre8, respectively.

If your project uses Kotlin reflection or testing facilities, you need to add the corresponding dependencies as well. The artifact IDs are kotlin-reflect for the reflection library, and kotlin-test and kotlin-test-junit for the testing libraries.

Compile Kotlin-only source code

To compile source code, specify the source directories in the <build> tag:

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

The Kotlin Maven Plugin needs to be referenced to compile the sources:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>

            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>

                <execution>
                    <id>test-compile</id>
                    <goals>
                        <goal>test-compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Starting from Kotlin 1.8.20, you can replace the whole <executions> element above with <extensions>true</extensions>. Enabling extensions automatically adds the compile, test-compile, kapt, and test-kapt executions to your build, bound to their appropriate lifecycle phases. If you need to configure an execution, you need to specify its ID. You can find an example of this in the next section.

> If several build plugins overwrite the default lifecycle and you also enabled the extensions option, the last plugin in the <build> section has priority in terms of lifecycle settings. All earlier changes to lifecycle settings are ignored.

Compile Kotlin and Java sources

To compile projects that include Kotlin and Java source code, invoke the Kotlin compiler before the Java compiler. In Maven terms it means that kotlin-maven-plugin should be run before maven-compiler-plugin using the following method, making sure that the kotlin plugin comes before the maven-compiler-plugin in your pom.xml file:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
```

```
            <version>${kotlin.version}</version>
            <extensions>true</extensions> <!-- You can set this option
            to automatically take information about lifecycles -->
            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal> <!-- You can skip the <goals> element
                        if you enable extensions for the plugin -->
                    </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
                            <sourceDir>${project.basedir}/src/main/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <goals> <goal>test-compile</goal> </goals> <!-- You can skip the <goals> element
                    if you enable extensions for the plugin -->
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/test/kotlin</sourceDir>
                            <sourceDir>${project.basedir}/src/test/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.5.1</version>
            <executions>
                <!-- Replacing default-compile as it is treated specially by Maven -->
                <execution>
                    <id>default-compile</id>
                    <phase>none</phase>
                </execution>
                <!-- Replacing default-testCompile as it is treated specially by Maven -->
                <execution>
                    <id>default-testCompile</id>
                    <phase>none</phase>
                </execution>
                <execution>
                    <id>java-compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>java-test-compile</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Incremental compilation

To make your builds faster, you can enable incremental compilation for Maven by defining the kotlin.compiler.incremental property:

```
<properties>
    <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>
```

Alternatively, run your build with the -Dkotlin.compiler.incremental=true option.

Annotation processing

See the description of Kotlin annotation processing tool (kapt).

Jar file

To create a small Jar file containing just the code from your module, include the following under build->plugins in your Maven pom.xml file, where main.class is defined as a property and points to the main Kotlin or Java class:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>${main.class}</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

Self-contained Jar file

To create a self-contained Jar file containing the code from your module along with dependencies, include the following under build->plugins in your Maven pom.xml file, where main.class is defined as a property and points to the main Kotlin or Java class:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.6</version>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals> <goal>single</goal> </goals>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>${main.class}</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This self-contained jar file can be passed directly to a JRE to run your application:

```
java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

Specifying compiler options

Additional options and arguments for the compiler can be specified as tags under the <configuration> element of the Maven plugin node:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <extensions>true</extensions> <!-- If you want to enable automatic addition of executions to your build -->
    <executions>...</executions>
    <configuration>
        <nowarn>true</nowarn>  <!-- Disable warnings -->
```

```
        <args>
            <arg>-Xjsr305=strict</arg> <!-- Enable strict mode for JSR-305 annotations -->
            ...
        </args>
    </configuration>
</plugin>
```

Many of the options can also be configured through properties:

```
<project ...>
    <properties>
        <kotlin.compiler.languageVersion>1.9</kotlin.compiler.languageVersion>
    </properties>
</project>
```

The following attributes are supported:

Attributes common to JVM and JS

Name	Property name	Description	Possible values	Default value
nowarn		Generate no warnings	true, false	false
languageVersion	kotlin.compiler.languageVersion	Provide source compatibility with the specified version of Kotlin	"1.3" (DEPRECATED), "1.4" (DEPRECATED), "1.5", "1.6", "1.7", "1.8", "1.9" (EXPERIMENTAL)	
apiVersion	kotlin.compiler.apiVersion	Allow using declarations only from the specified version of bundled libraries	"1.3" (DEPRECATED), "1.4" (DEPRECATED), "1.5", "1.6", "1.7", "1.8", "1.9" (EXPERIMENTAL)	
sourceDirs		The directories containing the source files to compile		The project source roots
compilerPlugins		Enabled compiler plugins		[]
pluginOptions		Options for compiler plugins		[]
args		Additional compiler arguments		[]

Attributes specific to JVM

Name	Property name	Description	Possible values	Default value
jvmTarget	kotlin.compiler.jvmTarget	Target version of the generated JVM bytecode	"1.8", "9", "10", ..., "20"	"1.8"
jdkHome	kotlin.compiler.jdkHome	Include a custom JDK from the specified location into the classpath instead of the default JAVA_HOME		

Attributes specific to JS

Name	Property name	Description	Possible values	Default value
outputFile		Destination *.js file for the compilation result		
metaInfo		Generate .meta.js and .kjsm files with metadata. Use to create a library	true, false	true
sourceMap		Generate source map	true, false	false
sourceMapEmbedSources		Embed source files into source map	"never", "always", "inlining"	"inlining"
sourceMapPrefix		Add the specified prefix to paths in the source map		
moduleKind		The kind of JS module generated by the compiler	"umd", "commonjs", "amd", "plain"	"umd"

Using BOM

To use a Kotlin Bill of Materials (BOM), write a dependency on kotlin-bom:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-bom</artifactId>
      <version>1.9.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Generating documentation

The standard Javadoc generation plugin (maven-javadoc-plugin) does not support Kotlin code. To generate documentation for Kotlin projects, use Dokka; please refer to the Dokka README for configuration instructions. Dokka supports mixed-language projects and can generate output in multiple formats, including standard Javadoc.

OSGi

For OSGi support see the Kotlin OSGi page.

Ant

Getting the Ant tasks

Kotlin provides three tasks for Ant:

- kotlinc: Kotlin compiler targeting the JVM
- kotlin2js: Kotlin compiler targeting JavaScript
- withKotlin: Task to compile Kotlin files when using the standard javac Ant task

These tasks are defined in the kotlin-ant.jar library which is located in the lib folder in the Kotlin Compiler archive. Ant version 1.8.2+ is required.

Targeting JVM with Kotlin-only source

When the project consists of exclusively Kotlin source code, the easiest way to compile the project is to use the kotlinc task:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc src="hello.kt" output="hello.jar"/>
    </target>
</project>
```

where ${kotlin.lib} points to the folder where the Kotlin standalone compiler was unzipped.

Targeting JVM with Kotlin-only source and multiple roots

If a project consists of multiple source roots, use src as elements to define paths:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="root1"/>
            <src path="root2"/>
        </kotlinc>
    </target>
</project>
```

Targeting JVM with Kotlin and Java source

If a project consists of both Kotlin and Java source code, while it is possible to use kotlinc, to avoid repetition of task parameters, it is recommended to use withKotlin task:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <delete dir="classes" failonerror="false"/>
        <mkdir dir="classes"/>
        <javac destdir="classes" includeAntRuntime="false" srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

You can also specify the name of the module being compiled as the moduleName attribute:

```
<withKotlin moduleName="myModule"/>
```

Targeting JavaScript with single source folder

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js"/>
    </target>
</project>
```

Targeting JavaScript with Prefix, PostFix and sourcemap options

```
<project name="Ant Task Test" default="build">
    <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js" outputPrefix="prefix" outputPostfix="postfix" sourcemap="true"/>
    </target>
</project>
```

Targeting JavaScript with single source folder and metaInfo option

The metaInfo option is useful, if you want to distribute the result of translation as a Kotlin/JavaScript library. If metaInfo was set to true, then during compilation additional JS file with binary metadata will be created. This file should be distributed together with the result of translation:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <!-- out.meta.js will be created, which contains binary metadata -->
        <kotlin2js src="root1" output="out.js" metaInfo="true"/>
    </target>
</project>
```

References

Complete list of elements and attributes are listed below:

Attributes common for kotlinc and kotlin2js

Name	Description	Required	Default Value
src	Kotlin source file or directory to compile	Yes	
nowarn	Suppresses all compilation warnings	No	false
noStdlib	Does not include the Kotlin standard library into the classpath	No	false
failOnError	Fails the build if errors are detected during the compilation	No	true

kotlinc attributes

Name	Description	Required	Default Value

Name	Description	Required	Default Value
output	Destination directory or .jar file name	Yes	
classpath	Compilation class path	No	
classpathref	Compilation class path reference	No	
includeRuntime	If output is a .jar file, whether Kotlin runtime library is included in the jar	No	true
moduleName	Name of the module being compiled	No	The name of the target (if specified) or the project

kotlin2js attributes

Name	Description	Required
output	Destination file	Yes
libraries	Paths to Kotlin libraries	No
outputPrefix	Prefix to use for generated JavaScript files	No
outputSuffix	Suffix to use for generated JavaScript files	No
sourcemap	Whether sourcemap file should be generated	No
metaInfo	Whether metadata file with binary descriptors should be generated	No
main	Should compiler generated code call the main function	No

Passing raw compiler arguments

To pass custom raw compiler arguments, you can use <compilerarg> elements with either value or line attributes. This can be done within the <kotlinc>, <kotlin2js>, and <withKotlin> task elements, as follows:

```
<kotlinc src="${test.data}/hello.kt" output="${temp}/hello.jar">
    <compilerarg value="-Xno-inline"/>
    <compilerarg line="-Xno-call-assertions -Xno-param-assertions"/>
    <compilerarg value="-Xno-optimize"/>
</kotlinc>
```

The full list of arguments that can be used is shown when you run kotlinc -help.

Introduction

Dokka is an API documentation engine for Kotlin.

Just like Kotlin itself, Dokka supports mixed-language projects. It understands Kotlin's KDoc comments and Java's Javadoc comments.

Dokka can generate documentation in multiple formats, including its own modern HTML format, multiple flavors of Markdown, and Java's Javadoc HTML.

Here are some libraries that use Dokka for their API reference documentation:

- kotlinx.coroutines
- Bitmovin
- Hexagon
- Ktor
- OkHttp (Markdown)

You can run Dokka using Gradle, Maven or from the command line. It is also highly pluggable.

See Get started with Dokka to take your first steps in using Dokka.

Community

Dokka has a dedicated #dokka channel in Kotlin Community Slack where you can chat about Dokka, its plugins and how to develop them, as well as get in touch with maintainers.

Get started with Dokka

Below you can find simple instructions to help you get started with Dokka.

Gradle Kotlin DSL

Apply the Gradle plugin for Dokka in the root build script of your project:

```
plugins {
    id("org.jetbrains.dokka") version "1.8.20"
}
```

When documenting multi-project builds, you need to apply the Gradle plugin within subprojects as well:

```
subprojects {
    apply(plugin = "org.jetbrains.dokka")
}
```

To generate documentation, run the following Gradle tasks:

- dokkaHtml for single-project builds
- dokkaHtmlMultiModule for multi-project builds

By default, the output directory is set to /build/dokka/html and /build/dokka/htmlMultiModule.

To learn more about using Dokka with Gradle, see Gradle.

Gradle Groovy DSL

Apply the Gradle plugin for Dokka in the root build script of your project:

```
plugins {
    id 'org.jetbrains.dokka' version '1.8.20'
}
```

When documenting multi-project builds, you need to apply the Gradle plugin within subprojects as well:

```
subprojects {
    apply plugin: 'org.jetbrains.dokka'
}
```

To generate documentation, run the following Gradle tasks:

- dokkaHtml for single-project builds
- dokkaHtmlMultiModule for multi-project builds

By default, the output directory is set to /build/dokka/html and /build/dokka/htmlMultiModule.

To learn more about using Dokka with Gradle, see Gradle.

Maven

Add the Maven plugin for Dokka to the plugins section of your POM file:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.dokka</groupId>
            <artifactId>dokka-maven-plugin</artifactId>
            <version>1.8.20</version>
            <executions>
                <execution>
                    <phase>pre-site</phase>
                    <goals>
                        <goal>dokka</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

To generate documentation, run the dokka:dokka goal.

By default, the output directory is set to target/dokka.

To learn more about using Dokka with Maven, see Maven.

Gradle

To generate documentation for a Gradle-based project, you can use the Gradle plugin for Dokka.

It comes with basic autoconfiguration for your project, has convenient Gradle tasks for generating documentation, and provides a great deal of configuration options to customize the output.

You can play around with Dokka and see how it can be configured for various projects by visiting our Gradle example projects.

Apply Dokka

The recommended way of applying the Gradle plugin for Dokka is with the plugins DSL:

Kotlin

```
plugins {
    id("org.jetbrains.dokka") version "1.8.20"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.dokka' version '1.8.20'
}
```

When documenting multi-project builds, you need to apply the Gradle plugin for Dokka within subprojects as well. You can use allprojects {} or subprojects {} Gradle configurations to achieve that:

Gradle Kotlin DSL

```kotlin
subprojects {
    apply(plugin = "org.jetbrains.dokka")
}
```

Gradle Groovy DSL

```groovy
subprojects {
    apply plugin: 'org.jetbrains.dokka'
}
```

See Configuration examples if you are not sure where to apply Dokka.

> Under the hood, Dokka uses the Kotlin Gradle plugin to perform autoconfiguration of source sets for which documentation is to be generated. Make sure to apply the Kotlin Gradle Plugin or configure source sets manually.

> If you are using Dokka in a precompiled script plugin, you need to add the Kotlin Gradle plugin as a dependency for it to work properly.

If you cannot use the plugins DSL for some reason, you can use the legacy method of applying plugins.

Generate documentation

The Gradle plugin for Dokka comes with HTML, Markdown and Javadoc output formats built in. It adds a number of tasks for generating documentation, both for single and multi-project builds.

Single-project builds

Use the following tasks to build documentation for simple, single-project applications and libraries:

Task	Description
dokkaHtml	Generates documentation in HTML format.

Experimental formats

Task	Description
dokkaGfm	Generates documentation in GitHub Flavored Markdown format.
dokkaJavadoc	Generates documentation in Javadoc format.
dokkaJekyll	Generates documentation in Jekyll compatible Markdown format.

By default, generated documentation is located in the build/dokka/{format} directory of your project. The output location, among other things, can be configured.

Multi-project builds

For documenting multi-project builds, make sure that you apply the Gradle plugin for Dokka within subprojects that you want to generate documentation for, as well as in their parent project.

MultiModule tasks

MultiModule tasks generate documentation for each subproject individually via Partial tasks, collect and process all outputs, and produce complete documentation with a common table of contents and resolved cross-project references.

Dokka creates the following tasks for parent projects automatically:

Task	Description
dokkaHtmlMultiModule	Generates multi-module documentation in HTML output format.

Experimental formats (multi-module)

Task	Description
dokkaGfmMultiModule	Generates multi-module documentation in GitHub Flavored Markdown output format.
dokkaJekyllMultiModule	Generates multi-module documentation in Jekyll compatible Markdown output format.

> The Javadoc output format does not have a MultiModule task, but a Collector task can be used instead.

By default, you can find ready-to-use documentation under {parentProject}/build/dokka/{format}MultiModule directory.

MultiModule results

Given a project with the following structure:

```
parentProject
    └── childProjectA
        ├── demo
            ├── ChildProjectAClass
    └── childProjectB
        ├── demo
            ├── ChildProjectBClass
```

These pages are generated after running dokkaHtmlMultiModule:

Screenshot for output of dokkaHtmlMultiModule task

See our multi-module project example for more details.

Collector tasks

Similar to MultiModule tasks, Collector tasks are created for each parent project: dokkaHtmlCollector, dokkaGfmCollector, dokkaJavadocCollector and dokkaJekyllCollector.

A Collector task executes the corresponding single-project task for each subproject (for example, dokkaHtml), and merges all outputs into a single virtual project.

The resulting documentation looks as if you have a single-project build that contains all declarations from the subprojects.

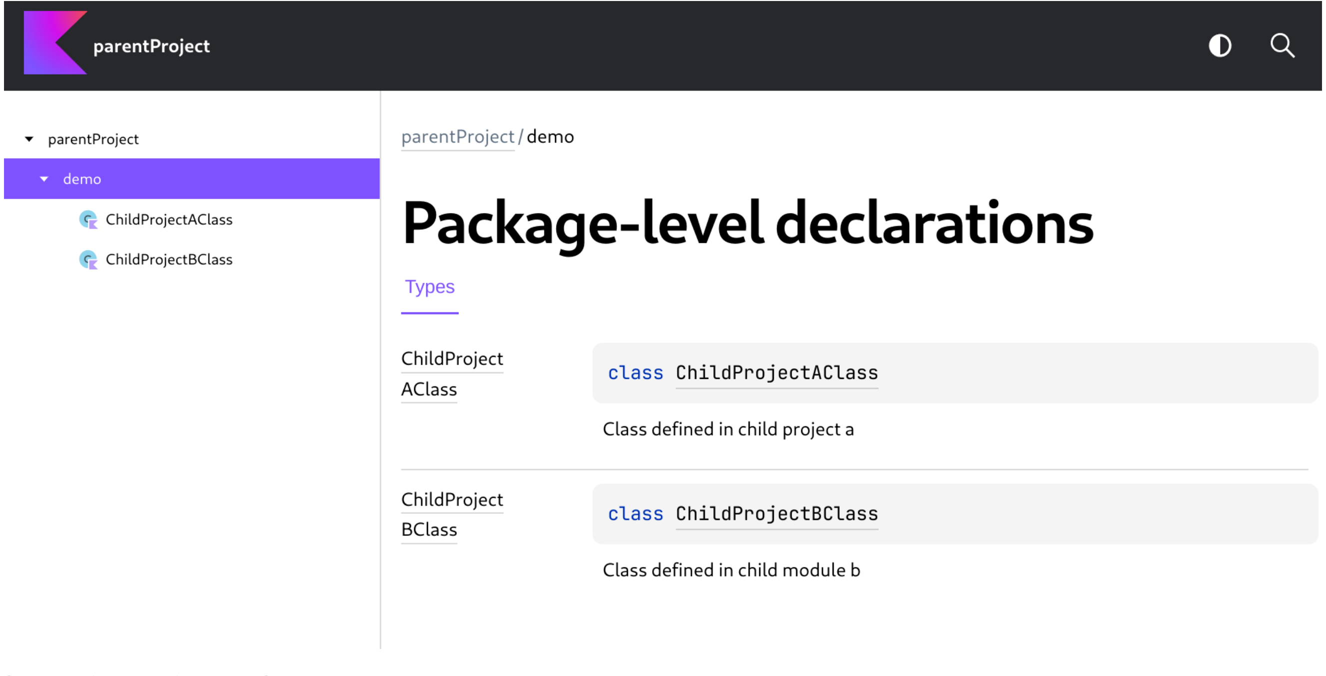
> Use the dokkaJavadocCollector task if you need to create Javadoc documentation for your multi-project build.

Collector results

Given a project with the following structure:

```
parentProject
    └── childProjectA
        ├── demo
            ├── ChildProjectAClass
    └── childProjectB
        ├── demo
            ├── ChildProjectBClass
```

These pages are generated after running dokkaHtmlCollector:



Screenshot for output of dokkaHtmlCollector task

See our multi-module project example for more details.

Partial tasks

Each subproject has Partial tasks created for it: dokkaHtmlPartial,dokkaGfmPartial, and dokkaJekyllPartial.

These tasks are not intended to be run independently, they are called by the parent's MultiModule task.

However, you can configure Partial tasks to customize Dokka for your subprojects.

> Output generated by Partial tasks contains unresolved HTML templates and references, so it cannot be used on its own without post-processing done by the parent's MultiModule task.

> If you want to generate documentation for a single subproject only, use single-project tasks. For example, :subprojectName:dokkaHtml.

Build javadoc.jar

If you want to publish your library to a repository, you may need to provide a javadoc.jar file that contains API reference documentation of your library.

For example, if you want to publish to Maven Central, you must supply a javadoc.jar alongside your project. However, not all repositories have that rule.

The Gradle plugin for Dokka does not provide any way to do this out of the box, but it can be achieved with custom Gradle tasks. One for generating documentation in HTML format and another one for Javadoc format:

Kotlin

```
tasks.register<Jar>("dokkaHtmlJar") {
    dependsOn(tasks.dokkaHtml)
    from(tasks.dokkaHtml.flatMap { it.outputDirectory })
    archiveClassifier.set("html-docs")
}

tasks.register<Jar>("dokkaJavadocJar") {
    dependsOn(tasks.dokkaJavadoc)
    from(tasks.dokkaJavadoc.flatMap { it.outputDirectory })
    archiveClassifier.set("javadoc")
```

```
}
```

Groovy

```
tasks.register('dokkaHtmlJar', Jar.class) {
    dependsOn(dokkaHtml)
    from(dokkaHtml)
    archiveClassifier.set("html-docs")
}

tasks.register('dokkaJavadocJar', Jar.class) {
    dependsOn(dokkaJavadoc)
    from(dokkaJavadoc)
    archiveClassifier.set("javadoc")
}
```

> If you publish your library to Maven Central, you can use services like javadoc.io to host your library's API documentation for free and without any setup. It takes documentation pages straight from the javadoc.jar. It works well with the HTML format as demonstrated in this example.

Configuration examples

Depending on the type of project that you have, the way you apply and configure Dokka differs slightly. However, configuration options themselves are the same, regardless of the type of your project.

For simple and flat projects with a single build.gradle.kts or build.gradle file found in the root of your project, see Single-project configuration.

For a more complex build with subprojects and multiple nested build.gradle.kts or build.gradle files, see Multi-project configuration.

Single-project configuration

Single-project builds usually have only one build.gradle.kts or build.gradle file in the root of the project, and typically have the following structure:

Kotlin

Single platform:

```
.
├── build.gradle.kts
└── src
    └── main
        └── kotlin
            └── HelloWorld.kt
```

Multiplatform:

```
.
├── build.gradle.kts
└── src
    └── commonMain
        └── kotlin
            └── Common.kt
    └── jvmMain
        └── kotlin
            └── JvmUtils.kt
    └── nativeMain
        └── kotlin
            └── NativeUtils.kt
```

Groovy

Single platform:

```
.
├── build.gradle
└── src
    └── main
        └── kotlin
```

```
            └── HelloWorld.kt
```

Multiplatform:

```
.
├── build.gradle
└── src
    └── commonMain
        └── kotlin
            └── Common.kt
    └── jvmMain
        └── kotlin
            └── JvmUtils.kt
    └── nativeMain
        └── kotlin
            └── NativeUtils.kt
```

In such projects, you need to apply Dokka and its configuration in the root build.gradle.kts or build.gradle file.

You can configure tasks and output formats individually:

Kotlin

Inside ./build.gradle.kts:

```kotlin
plugins {
    id("org.jetbrains.dokka") version "1.8.20"
}

tasks.dokkaHtml {
    outputDirectory.set(buildDir.resolve("documentation/html"))
}

tasks.dokkaGfm {
    outputDirectory.set(buildDir.resolve("documentation/markdown"))
}
```

Groovy

Inside ./build.gradle:

```groovy
plugins {
    id 'org.jetbrains.dokka' version '1.8.20'
}

dokkaHtml {
    outputDirectory.set(file("build/documentation/html"))
}

dokkaGfm {
    outputDirectory.set(file("build/documentation/markdown"))
}
```

Or you can configure all tasks and output formats at the same time:

Kotlin

Inside ./build.gradle.kts:

```kotlin
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.gradle.DokkaTaskPartial
import org.jetbrains.dokka.DokkaConfiguration.Visibility

plugins {
    id("org.jetbrains.dokka") version "1.8.20"
}

// Configure all single-project Dokka tasks at the same time,
// such as dokkaHtml, dokkaJavadoc and dokkaGfm.
tasks.withType<DokkaTask>().configureEach {
    dokkaSourceSets.configureEach {
```

```
        documentedVisibilities.set(
            setOf(
                Visibility.PUBLIC,
                Visibility.PROTECTED,
            )
        )

        perPackageOption {
            matchingRegex.set(".*internal.*")
            suppress.set(true)
        }
    }
}
```

Groovy

Inside ./build.gradle:

```
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.gradle.DokkaTaskPartial
import org.jetbrains.dokka.DokkaConfiguration.Visibility

plugins {
    id 'org.jetbrains.dokka' version '1.8.20'
}

// Configure all single-project Dokka tasks at the same time,
// such as dokkaHtml, dokkaJavadoc and dokkaGfm.
tasks.withType(DokkaTask.class) {
    dokkaSourceSets.configureEach {
        documentedVisibilities.set([
                Visibility.PUBLIC,
                Visibility.PROTECTED
        ])

        perPackageOption {
            matchingRegex.set(".*internal.*")
            suppress.set(true)
        }
    }
}
```

Multi-project configuration

Gradle's multi-project builds are more complex in structure and configuration. They usually have multiple nested build.gradle.kts or build.gradle files, and typically have the following structure:

Kotlin

```
.
├── build.gradle.kts
├── settings.gradle.kts
├── subproject-A
    └── build.gradle.kts
    └── src
        └── main
            └── kotlin
                └── HelloFromA.kt
├── subproject-B
    └── build.gradle.kts
    └── src
        └── main
            └── kotlin
                └── HelloFromB.kt
```

Groovy

```
.
├── build.gradle
├── settings.gradle
├── subproject-A
    └── build.gradle
    └── src
        └── main
```

```
            └── kotlin
                └── HelloFromA.kt
├── subproject-B
    └── build.gradle
    └── src
        └── main
            └── kotlin
                └── HelloFromB.kt
```

In this case, there are multiple ways of applying and configuring Dokka.

Subproject configuration

To configure subprojects in a multi-project build, you need to configure Partial tasks.

You can configure all subprojects at the same time in the root build.gradle.kts or build.gradle file, using Gradle's allprojects {} or subprojects {} configuration blocks:

Kotlin

In the root ./build.gradle.kts:

```
import org.jetbrains.dokka.gradle.DokkaTaskPartial

plugins {
    id("org.jetbrains.dokka") version "1.8.20"
}

subprojects {
    apply(plugin = "org.jetbrains.dokka")

    // configure only the HTML task
    tasks.dokkaHtmlPartial {
        outputDirectory.set(buildDir.resolve("docs/partial"))
    }

    // configure all format tasks at once
    tasks.withType<DokkaTaskPartial>().configureEach {
        dokkaSourceSets.configureEach {
            includes.from("README.md")
        }
    }
}
```

Groovy

In the root ./build.gradle:

```
import org.jetbrains.dokka.gradle.DokkaTaskPartial

plugins {
    id 'org.jetbrains.dokka' version '1.8.20'
}

subprojects {
    apply plugin: 'org.jetbrains.dokka'

    // configure only the HTML task
    dokkaHtmlPartial {
        outputDirectory.set(file("build/docs/partial"))
    }

    // configure all format tasks at once
    tasks.withType(DokkaTaskPartial.class) {
        dokkaSourceSets.configureEach {
            includes.from("README.md")
        }
    }
}
```

Alternatively, you can apply and configure Dokka within subprojects individually.

For example, to have specific settings for the subproject-A subproject only, you need to apply the following code inside ./subproject-A/build.gradle.kts:

Kotlin

Inside ./subproject-A/build.gradle.kts:

```kotlin
apply(plugin = "org.jetbrains.dokka")

// configuration for subproject-A only.
tasks.dokkaHtmlPartial {
    outputDirectory.set(buildDir.resolve("docs/partial"))
}
```

Groovy

Inside ./subproject-A/build.gradle:

```groovy
apply plugin: 'org.jetbrains.dokka'

// configuration for subproject-A only.
dokkaHtmlPartial {
    outputDirectory.set(file("build/docs/partial"))
}
```

Parent project configuration

If you want to configure something which is universal across all documentation and does not belong to the subprojects - in other words, it's a property of the parent project - you need to configure the MultiModule tasks.

For example, if you want to change the name of your project which is used in the header of the HTML documentation, you need to apply the following inside the root build.gradle.kts or build.gradle file:

Kotlin

In the root ./build.gradle.kts file:

```kotlin
plugins {
    id("org.jetbrains.dokka") version "1.8.20"
}

tasks.dokkaHtmlMultiModule {
    moduleName.set("WHOLE PROJECT NAME USED IN THE HEADER")
}
```

Groovy

In the root ./build.gradle file:

```groovy
plugins {
    id 'org.jetbrains.dokka' version '1.8.20'
}

dokkaHtmlMultiModule {
    moduleName.set("WHOLE PROJECT NAME USED IN THE HEADER")
}
```

Configuration options

Dokka has many configuration options to tailor your and your reader's experience.

Below are some examples and detailed descriptions for each configuration section. You can also find an example with all configuration options applied at the bottom of the page.

See Configuration examples for more details on where to apply configuration blocks and how.

General configuration

Here is an example of general configuration of any Dokka task, regardless of source set or package:

Kotlin

```kotlin
import org.jetbrains.dokka.gradle.DokkaTask

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(buildDir.resolve("dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    // ..
    // source set configuration section
    // ..
}
```

Groovy

```groovy
import org.jetbrains.dokka.gradle.DokkaTask

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(file("build/dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    // ..
    // source set configuration section
    // ..
}
```

moduleName

The display name used to refer to the module. It is used for the table of contents, navigation, logging, etc.

If set for a single-project build or a MultiModule task, it is used as the project name.

Default: Gradle project name

moduleVersion

The module version. If set for a single-project build or a MultiModule task, it is used as the project version.

Default: Gradle project version

outputDirectory

The directory to where documentation is generated, regardless of format. It can be set on a per-task basis.

The default is {project}/{buildDir}/{format}, where {format} is the task name with the "dokka" prefix removed. For the dokkaHtmlMultiModule task, it is project/buildDir/htmlMultiModule.

failOnWarning

Whether to fail documentation generation if Dokka has emitted a warning or an error. The process waits until all errors and warnings have been emitted first.

This setting works well with reportUndocumented.

Default: false

suppressObviousFunctions

Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from kotlin.Any, Kotlin.Enum, java.lang.Object or java.lang.Enum, such as equals, hashCode, toString.

- Synthetic (generated by the compiler) and does not have any documentation, such as dataClass.componentN or dataClass.copy.

Default: true

suppressInheritedMembers

Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This can suppress functions such as equals / hashCode / toString, but cannot suppress synthetic functions such as dataClass.componentN and dataClass.copy. Use suppressObviousFunctions for that.

Default: false

offlineMode

Whether to resolve remote files/links over your network.

This includes package-lists used for generating external documentation links. For example, to make classes from the standard library clickable.

Setting this to true can significantly speed up build times in certain cases, but can also worsen documentation quality and user experience. For example, by not resolving class/member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See externalDocumentationLinks section.

Default: false

Source set configuration

Dokka allows configuring some options for Kotlin source sets:

Kotlin

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets {
        // configuration exclusive to the 'linux' source set
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md", "extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(), file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                // Source link section
            }
            externalDocumentationLink {
                // External documentation link section
            }
            perPackageOption {
```

```
                // Package options section
            }
        }
    }
}
```

Groovy

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets {
        // configuration exclusive to the 'linux' source set
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set([Visibility.PUBLIC])
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md", "extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(), file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                // Source link section
            }
            externalDocumentationLink {
                // External documentation link section
            }
            perPackageOption {
                // Package options section
            }
        }
    }
}
```

suppress

Whether this source set should be skipped when generating documentation.

Default: false

displayName

The display name used to refer to this source set.

The name is used both externally (for example, as source set name visible to documentation readers) and internally (for example, for logging messages of reportUndocumented).

By default, the value is deduced from information provided by the Kotlin Gradle plugin.

documentedVisibilities

The set of visibility modifiers that should be documented.

This can be used if you want to document protected/internal/private declarations, as well as if you want to exclude public declarations and only document internal

API.

This can be configured on per-package basis.

Default: DokkaConfiguration.Visibility.PUBLIC

reportUndocumented

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by documentedVisibilities and other filters.

This setting works well with failOnWarning.

This can be configured on per-package basis.

Default: false

skipEmptyPackages

Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if skipDeprecated is set to true and your package contains only deprecated declarations, it is considered to be empty.

Default: true

skipDeprecated

Whether to document declarations annotated with @Deprecated.

This can be configured on per-package basis.

Default: false

suppressGeneratedFiles

Whether to document/analyze generated files.

Generated files are expected to be present under the {project}/{buildDir}/generated directory.

If set to true, it effectively adds all files from that directory to the suppressedFiles option, so you can configure it manually.

Default: true

jdkVersion

The JDK version to use when generating external documentation links for Java types.

For example, if you use java.util.UUID in some public declaration signature, and this option is set to 8, Dokka generates an external documentation link to JDK 8 Javadocs for it.

Default: JDK 8

languageVersion

The Kotlin language version used for setting up analysis and @sample environment.

By default, the latest language version available to Dokka's embedded compiler is used.

apiVersion

The Kotlin API version used for setting up analysis and @sample environment.

By default, it is deduced from languageVersion.

noStdlibLink

Whether to generate external documentation links that lead to the API reference documentation of Kotlin's standard library.

Note: Links are generated when noStdLibLink is set to false.

Default: false

noJdkLink

Whether to generate external documentation links to JDK's Javadocs.

The version of JDK Javadocs is determined by the jdkVersion option.

Note: Links are generated when noJdkLink is set to false.

Default: false

noAndroidSdkLink

Whether to generate external documentation links to the Android SDK API reference.

This is only relevant in Android projects, ignored otherwise.

Note: Links are generated when noAndroidSdkLink is set to false.

Default: false

includes

A list of Markdown files that contain module and package documentation.

The contents of the specified files are parsed and embedded into documentation as module and package descriptions.

See Dokka gradle example for an example of what it looks like and how to use it.

platform

The platform to be used for setting up code analysis and @sample environment.

The default value is deduced from information provided by the Kotlin Gradle plugin.

sourceRoots

The source code roots to be analyzed and documented. Acceptable inputs are directories and individual .kt / .java files.

By default, source roots are deduced from information provided by the Kotlin Gradle plugin.

classpath

The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved/picked up automatically.

This option accepts both .jar and .klib files.

By default, classpath is deduced from information provided by the Kotlin Gradle plugin.

samples

A list of directories or files that contain sample functions which are referenced via the @sample KDoc tag.

Source link configuration

The sourceLinks configuration block allows you to add a source link to each signature that leads to the remoteUrl with a specific line number. (The line number is configurable by setting remoteLineSuffix).

This helps readers to find the source code for each declaration.

For an example, see the documentation for the count() function in kotlinx.coroutines.

Kotlin

```kotlin
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        sourceLink {
            localDirectory.set(projectDir.resolve("src"))
            remoteUrl.set(URL("https://github.com/kotlin/dokka/tree/master/src"))
            remoteLineSuffix.set("#L")
        }
    }
}
```

Groovy

```groovy
import org.jetbrains.dokka.gradle.DokkaTask
```

```
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        sourceLink {
            localDirectory.set(file("src"))
            remoteUrl.set(new URL("https://github.com/kotlin/dokka/tree/master/src"))
            remoteLineSuffix.set("#L")
        }
    }
}
```

localDirectory

The path to the local source directory. The path must be relative to the root of the current project.

remoteUrl

The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, etc. This URL is used to generate source code links of declarations.

remoteLineSuffix

The suffix used to append the source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to #L and the line number is 10, the resulting URL suffix is #L10.

Suffixes used by popular services:

- GitHub: #L
- GitLab: #L
- Bitbucket: #lines-

Default: #L

Package options

The perPackageOption configuration block allows setting some options for specific packages matched by matchingRegex.

Kotlin

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        perPackageOption {
            matchingRegex.set(".*api.*")
            suppress.set(false)
            skipDeprecated.set(false)
            reportUndocumented.set(false)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
        }
    }
```

```
}
```

Groovy

```groovy
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // Source set configuration section
        // ..

        perPackageOption {
            matchingRegex.set(".*api.*")
            suppress.set(false)
            skipDeprecated.set(false)
            reportUndocumented.set(false)
            documentedVisibilities.set([Visibility.PUBLIC])
        }
    }
}
```

matchingRegex

The regular expression that is used to match the package.

Default: .*

suppress

Whether this package should be skipped when generating documentation.

Default: false

skipDeprecated

Whether to document declarations annotated with @Deprecated.

This can be configured on source set level.

Default: false

reportUndocumented

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by documentedVisibilities and other filters.

This setting works well with failOnWarning.

This can be configured on source set level.

Default: false

documentedVisibilities

The set of visibility modifiers that should be documented.

This can be used if you want to document protected/internal/private declarations within this package, as well as if you want to exclude public declarations and only document internal API.

This can be configured on source set level.

Default: DokkaConfiguration.Visibility.PUBLIC

External documentation links configuration

The externalDocumentationLink block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from kotlinx.serialization, by default they are unclickable in your documentation, as if they are unresolved. However, since the API reference documentation for kotlinx.serialization is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. Thus

allowing Dokka to generate links for types from the library, making them resolve successfully and clickable.

By default, external documentation links for Kotlin standard library, JDK, Android SDK and AndroidX are configured.

Kotlin

```kotlin
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        externalDocumentationLink {
            url.set(URL("https://kotlinlang.org/api/kotlinx.serialization/"))
            packageListUrl.set(
                rootProject.projectDir.resolve("serialization.package.list").toURL()
            )
        }
    }
}
```

Groovy

```groovy
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        externalDocumentationLink {
            url.set(new URL("https://kotlinlang.org/api/kotlinx.serialization/"))
            packageListUrl.set(
                file("serialization.package.list").toURL()
            )
        }
    }
}
```

url

The root URL of documentation to link to. It must contain a trailing slash.

Dokka does its best to automatically find package-list for the given URL, and link declarations together.

If automatic resolution fails or if you want to use locally cached files instead, consider setting the packageListUrl option.

packageListUrl

The exact location of a package-list. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as module and package names.

This can also be a locally cached file to avoid network calls.

Complete configuration

Below you can see all possible configuration options applied at the same time.

Kotlin

```kotlin
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(buildDir.resolve("dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    dokkaSourceSets {
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md", "extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(), file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                localDirectory.set(projectDir.resolve("src"))
                remoteUrl.set(URL("https://github.com/kotlin/dokka/tree/master/src"))
                remoteLineSuffix.set("#L")
            }

            externalDocumentationLink {
                url.set(URL("https://kotlinlang.org/api/latest/jvm/stdlib/"))
                packageListUrl.set(
                    rootProject.projectDir.resolve("stdlib.package.list").toURL()
                )
            }

            perPackageOption {
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
                documentedVisibilities.set(
                    setOf(
                        Visibility.PUBLIC,
                        Visibility.PRIVATE,
                        Visibility.PROTECTED,
                        Visibility.INTERNAL,
                        Visibility.PACKAGE
                    )
                )
            }
        }
    }
}
```

Groovy

```groovy
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(file("build/dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    dokkaSourceSets {
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set([Visibility.PUBLIC])
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md", "extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(), file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                localDirectory.set(file("src"))
                remoteUrl.set(new URL("https://github.com/kotlin/dokka/tree/master/src"))
                remoteLineSuffix.set("#L")
            }

            externalDocumentationLink {
                url.set(new URL("https://kotlinlang.org/api/latest/jvm/stdlib/"))
                packageListUrl.set(
                        file("stdlib.package.list").toURL()
                )
            }

            perPackageOption {
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
                documentedVisibilities.set([Visibility.PUBLIC])
            }
        }
    }
}
```

Maven

To generate documentation for a Maven-based project, you can use the Maven plugin for Dokka.

> Compared to the Gradle plugin for Dokka, the Maven plugin has only basic features and does not provide support for multi-module builds.

You can play around with Dokka and see how it can be configured for a Maven project by visiting our Maven example project.

Apply Dokka

To apply Dokka, you need to add dokka-maven-plugin to the plugins section of your POM file:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.dokka</groupId>
            <artifactId>dokka-maven-plugin</artifactId>
            <version>1.8.20</version>
            <executions>
                <execution>
                    <phase>pre-site</phase>
                    <goals>
                        <goal>dokka</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Generate documentation

The following goals are provided by the Maven plugin:

Goal	Description
dokka:dokka	Generates documentation with Dokka plugins applied. HTML format by default.

Experimental

Goal	Description
dokka:javadoc	Generates documentation in Javadoc format.
dokka:javadocJar	Generates a javadoc.jar file that contains documentation in Javadoc format.

Other output formats

By default, the Maven plugin for Dokka builds documentation in HTML output format.

All other output formats are implemented as Dokka plugins. In order to generate documentation in the desired format, you have to add it as a Dokka plugin to the configuration.

For example, to use the experimental GFM format, you have to add gfm-plugin artifact:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>gfm-plugin</artifactId>
                <version>1.8.20</version>
            </plugin>
```

```
        </dokkaPlugins>
    </configuration>
</plugin>
```

With this configuration, running the dokka:dokka goal produces documentation in GFM format.

To learn more about Dokka plugins, see Dokka plugins.

Build javadoc.jar

If you want to publish your library to a repository, you may need to provide a javadoc.jar file that contains API reference documentation of your library.

For example, if you want to publish to Maven Central, you must supply a javadoc.jar alongside your project. However, not all repositories have that rule.

Unlike the Gradle plugin for Dokka, the Maven plugin comes with a ready-to-use dokka:javadocJar goal. By default, it generates documentation in Javadoc output format in thetarget folder.

If you are not satisfied with the built-in goal or want to customize the output (for example, you want to generate documentation in HTML format instead of Javadoc), similar behavior can be achieved by adding the Maven JAR plugin with the following configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.3.0</version>
    <executions>
        <execution>
            <goals>
                <goal>test-jar</goal>
            </goals>
        </execution>
        <execution>
            <id>dokka-jar</id>
            <phase>package</phase>
            <goals>
                <goal>jar</goal>
            </goals>
            <configuration>
                <classifier>dokka</classifier>
                <classesDirectory>${project.build.directory}/dokka</classesDirectory>
                <skipIfEmpty>true</skipIfEmpty>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The documentation and the .jar archive for it are then generated by running dokka:dokka and jar:jar@dokka-jar goals:

```
mvn dokka:dokka jar:jar@dokka-jar
```

> If you publish your library to Maven Central, you can use services like javadoc.io to host your library's API documentation for free and without any setup. It takes documentation pages straight from the javadoc.jar. It works well with the HTML format as demonstrated in this example.

Configuration example

Maven's plugin configuration block can be used to configure Dokka.

Here is an example of a basic configuration that only changes the output location of your documentation:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <outputDir>${project.basedir}/target/documentation/dokka</outputDir>
    </configuration>
</plugin>
```

Configuration options

Dokka has many configuration options to tailor your and your reader's experience.

Below are some examples and detailed descriptions for each configuration section. You can also find an example with all configuration options applied at the bottom of the page.

General configuration

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <skip>false</skip>
        <moduleName>${project.artifactId}</moduleName>
        <outputDir>${project.basedir}/target/documentation</outputDir>
        <failOnWarning>false</failOnWarning>
        <suppressObviousFunctions>true</suppressObviousFunctions>
        <suppressInheritedMembers>false</suppressInheritedMembers>
        <offlineMode>false</offlineMode>
        <sourceDirectories>
            <dir>${project.basedir}/src</dir>
        </sourceDirectories>
        <documentedVisibilities>
            <visibility>PUBLIC</visibility>
            <visibility>PROTECTED</visibility>
        </documentedVisibilities>
        <reportUndocumented>false</reportUndocumented>
        <skipDeprecated>false</skipDeprecated>
        <skipEmptyPackages>true</skipEmptyPackages>
        <suppressedFiles>
            <file>/path/to/dir</file>
            <file>/path/to/file</file>
        </suppressedFiles>
        <jdkVersion>8</jdkVersion>
        <languageVersion>1.7</languageVersion>
        <apiVersion>1.7</apiVersion>
        <noStdlibLink>false</noStdlibLink>
        <noJdkLink>false</noJdkLink>
        <includes>
            <include>packages.md</include>
            <include>extra.md</include>
        </includes>
        <classpath>${project.compileClasspathElements}</classpath>
        <samples>
            <dir>${project.basedir}/samples</dir>
        </samples>
        <sourceLinks>
            <!-- Separate section -->
        </sourceLinks>
        <externalDocumentationLinks>
            <!-- Separate section -->
        </externalDocumentationLinks>
        <perPackageOptions>
            <!-- Separate section -->
        </perPackageOptions>
    </configuration>
</plugin>
```

skip
Whether to skip documentation generation.

Default: false

moduleName
The display name used to refer to the project/module. It's used for the table of contents, navigation, logging, etc.

Default: {project.artifactId}

outputDir
The directory to where documentation is generated, regardless of format.

Default: {project.basedir}/target/dokka

failOnWarning
Whether to fail documentation generation if Dokka has emitted a warning or an error. The process waits until all errors and warnings have been emitted first.

This setting works well with reportUndocumented.

Default: false

suppressObviousFunctions

Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from kotlin.Any, Kotlin.Enum, java.lang.Object or java.lang.Enum, such as equals, hashCode, toString.
- Synthetic (generated by the compiler) and does not have any documentation, such as dataClass.componentN or dataClass.copy.

Default: true

suppressInheritedMembers

Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This can suppress functions such as equals/hashCode/toString, but cannot suppress synthetic functions such as dataClass.componentN and dataClass.copy. Use suppressObviousFunctions for that.

Default: false

offlineMode

Whether to resolve remote files/links over your network.

This includes package-lists used for generating external documentation links. For example, to make classes from the standard library clickable.

Setting this to true can significantly speed up build times in certain cases, but can also worsen documentation quality and user experience. For example, by not resolving class/member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See externalDocumentationLinks section.

Default: false

sourceDirectories

The source code roots to be analyzed and documented. Acceptable inputs are directories and individual .kt / .java files.

Default: {project.compileSourceRoots}

documentedVisibilities

The set of visibility modifiers that should be documented.

This can be used if you want to document protected/internal/private declarations, as well as if you want to exclude public declarations and only document internal API.

Can be configured on per-package basis.

Default: PUBLIC

reportUndocumented

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by documentedVisibilities and other filters.

This setting works well with failOnWarning.

This can be overridden at package level.

Default: false

skipDeprecated

Whether to document declarations annotated with @Deprecated.

This can be overridden at package level.

Default: false

skipEmptyPackages

Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if skipDeprecated is set to true and your package contains only deprecated declarations, it is considered to be empty.

Default: true

suppressedFiles

The directories or individual files that should be suppressed, meaning that declarations from them are not documented.

jdkVersion

The JDK version to use when generating external documentation links for Java types.

For example, if you use java.util.UUID in some public declaration signature, and this option is set to 8, Dokka generates an external documentation link to JDK 8 Javadocs for it.

Default: JDK 8

languageVersion

The Kotlin language version used for setting up analysis and @sample environment.

By default, the latest language version available to Dokka's embedded compiler is used.

apiVersion

The Kotlin API version used for setting up analysis and @sample environment.

By default, it is deduced from languageVersion.

noStdlibLink

Whether to generate external documentation links that lead to the API reference documentation of Kotlin's standard library.

Note: Links are generated when noStdLibLink is set to false.

Default: false

noJdkLink

Whether to generate external documentation links to JDK's Javadocs.

The version of JDK Javadocs is determined by the jdkVersion option.

Note: Links are generated when noJdkLink is set to false.

Default: false

includes

A list of Markdown files that contain module and package documentation

The contents of specified files are parsed and embedded into documentation as module and package descriptions.

classpath

The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved/picked up automatically. This option accepts both .jar and .klib files.

Default: {project.compileClasspathElements}

samples

A list of directories or files that contain sample functions which are referenced via @sample KDoc tag.

Source link configuration

The sourceLinks configuration block allows you to add a source link to each signature that leads to the url with a specific line number. (The line number is configurable by setting lineSuffix).

This helps readers to find the source code for each declaration.

For an example, see the documentation for the count() function in kotlinx.coroutines.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <sourceLinks>
            <link>
                <path>${project.basedir}/src</path>
                <url>https://github.com/kotlin/dokka/tree/master/src</url>
                <lineSuffix>#L</lineSuffix>
            </link>
        </sourceLinks>
    </configuration>
</plugin>
```

path
The path to the local source directory. The path must be relative to the root of the current module.

url
The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, etc. This URL is used to generate source code links of declarations.

lineSuffix
The suffix used to append source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to #L and the line number is 10, the resulting URL suffix is #L10.

Suffixes used by popular services:

- GitHub: #L
- GitLab: #L
- Bitbucket: #lines-

External documentation links configuration

The externalDocumentationLinks block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from kotlinx.serialization, by default they are unclickable in your documentation, as if they are unresolved. However, since the API reference documentation for kotlinx.serialization is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. Thus allowing Dokka to generate links for types from the library, making them resolve successfully and clickable.

By default, external documentation links for Kotlin standard library and JDK are configured.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <externalDocumentationLinks>
            <link>
                <url>https://kotlinlang.org/api/kotlinx.serialization/</url>
                <packageListUrl>file:/${project.basedir}/serialization.package.list</packageListUrl>
            </link>
        </externalDocumentationLinks>
    </configuration>
</plugin>
```

url
The root URL of documentation to link to. It must contain a trailing slash.

Dokka does its best to automatically find the package-list for the given URL, and link declarations together.

If automatic resolution fails or if you want to use locally cached files instead, consider setting the packageListUrl option.

packageListUrl
The exact location of a package-list. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as module and package names.

This can also be a locally cached file to avoid network calls.

Package options

The perPackageOptions configuration block allows setting some options for specific packages matched by matchingRegex.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <perPackageOptions>
            <packageOptions>
                <matchingRegex>.*api.*</matchingRegex>
                <suppress>false</suppress>
                <reportUndocumented>false</reportUndocumented>
                <skipDeprecated>false</skipDeprecated>
```

```
                <documentedVisibilities>
                    <visibility>PUBLIC</visibility>
                    <visibility>PRIVATE</visibility>
                    <visibility>PROTECTED</visibility>
                    <visibility>INTERNAL</visibility>
                    <visibility>PACKAGE</visibility>
                </documentedVisibilities>
            </packageOptions>
        </perPackageOptions>
    </configuration>
</plugin>
```

matchingRegex
The regular expression that is used to match the package.

Default: .*

suppress
Whether this package should be skipped when generating documentation.

Default: false

documentedVisibilities
The set of visibility modifiers that should be documented.

This can be used if you want to document protected/internal/private declarations within this package, as well as if you want to exclude public declarations and only document internal API.

Default: PUBLIC

skipDeprecated
Whether to document declarations annotated with @Deprecated.

This can be set on project/module level.

Default: false

reportUndocumented
Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by documentedVisibilities and other filters.

This setting works well with failOnWarning.

Default: false

Complete configuration

Below you can see all the possible configuration options applied at the same time.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!-- ... -->
    <configuration>
        <skip>false</skip>
        <moduleName>${project.artifactId}</moduleName>
        <outputDir>${project.basedir}/target/documentation</outputDir>
        <failOnWarning>false</failOnWarning>
        <suppressObviousFunctions>true</suppressObviousFunctions>
        <suppressInheritedMembers>false</suppressInheritedMembers>
        <offlineMode>false</offlineMode>
        <sourceDirectories>
            <dir>${project.basedir}/src</dir>
        </sourceDirectories>
        <documentedVisibilities>
            <visibility>PUBLIC</visibility>
            <visibility>PRIVATE</visibility>
            <visibility>PROTECTED</visibility>
            <visibility>INTERNAL</visibility>
            <visibility>PACKAGE</visibility>
        </documentedVisibilities>
        <reportUndocumented>false</reportUndocumented>
        <skipDeprecated>false</skipDeprecated>
        <skipEmptyPackages>true</skipEmptyPackages>
        <suppressedFiles>
            <file>/path/to/dir</file>
```

```xml
            <file>/path/to/file</file>
        </suppressedFiles>
        <jdkVersion>8</jdkVersion>
        <languageVersion>1.7</languageVersion>
        <apiVersion>1.7</apiVersion>
        <noStdlibLink>false</noStdlibLink>
        <noJdkLink>false</noJdkLink>
        <includes>
            <include>packages.md</include>
            <include>extra.md</include>
        </includes>
        <classpath>${project.compileClasspathElements}</classpath>
        <samples>
            <dir>${project.basedir}/samples</dir>
        </samples>
        <sourceLinks>
            <link>
                <path>${project.basedir}/src</path>
                <url>https://github.com/kotlin/dokka/tree/master/src</url>
                <lineSuffix>#L</lineSuffix>
            </link>
        </sourceLinks>
        <externalDocumentationLinks>
            <link>
                <url>https://kotlinlang.org/api/latest/jvm/stdlib/</url>
                <packageListUrl>file:/${project.basedir}/stdlib.package.list</packageListUrl>
            </link>
        </externalDocumentationLinks>
        <perPackageOptions>
            <packageOptions>
                <matchingRegex>.*api.*</matchingRegex>
                <suppress>false</suppress>
                <reportUndocumented>false</reportUndocumented>
                <skipDeprecated>false</skipDeprecated>
                <documentedVisibilities>
                    <visibility>PUBLIC</visibility>
                    <visibility>PRIVATE</visibility>
                    <visibility>PROTECTED</visibility>
                    <visibility>INTERNAL</visibility>
                    <visibility>PACKAGE</visibility>
                </documentedVisibilities>
            </packageOptions>
        </perPackageOptions>
    </configuration>
</plugin>
```

CLI

If for some reason you cannot use Gradle or Maven build tools, Dokka has a command line (CLI) runner for generating documentation.

In comparison, it has the same, if not more, capabilities as the Gradle plugin for Dokka. Although it is considerably more difficult to set up as there is no autoconfiguration, especially in multiplatform and multi-module environments.

Get started

The CLI runner is published to Maven Central as a separate runnable artifact.

You can find it on mvnrepository or by browsing maven central repository directories directly.

With the dokka-cli-1.8.20.jar file saved on your computer, run it with the -help option to see all available configuration options and their description:

```
java -jar dokka-cli-1.8.20.jar -help
```

It also works for some nested options, such as -sourceSet:

```
java -jar dokka-cli-1.8.20.jar -sourceSet -help
```

Generate documentation

Prerequisites

Since there is no build tool to manage dependencies, you have to provide dependency .jar files yourself.

Listed below are the dependencies that you need for any output format:

Group	Artifact	Version	Link
org.jetbrains.dokka	dokka-base	1.8.20	mvnrepository
org.jetbrains.dokka	dokka-analysis	1.8.20	mvnrepository
org.jetbrains.dokka	kotlin-analysis-compiler	1.8.20	mvnrepository
org.jetbrains.dokka	kotlin-analysis-intellij	1.8.20	mvnrepository

Below are the additional dependencies that you need for HTML output format:

Group	Artifact	Version	Link
org.jetbrains.kotlinx	kotlinx-html-jvm	0.8.0	mvnrepository
org.freemarker	freemarker	2.3.31	mvnrepository

Run with command line options

You can pass command line options to configure the CLI runner.

At the very least you need to provide the following options:

- -pluginsClasspath - a list of absolute/relative paths to downloaded dependencies, separated by semi-colons ;
- -sourceSet - an absolute path to code sources to generate documentation for
- -outputDir - an absolute/relative path of the documentation output directory

```
java -jar dokka-cli-1.8.20.jar \
     -pluginsClasspath "./dokka-base-1.8.20.jar;./dokka-analysis-1.8.20.jar;./kotlin-analysis-intellij-1.8.20.jar;./kotlin-analysis-compiler-1.8.20.jar;./kotlinx-html-jvm-0.8.0.jar;./freemarker-2.3.31.jar" \
     -sourceSet "-src /home/myCoolProject/src/main/kotlin" \
     -outputDir "./dokka/html"
```

> Due to an internal class conflict, first pass kotlin-analysis-intellij and only then kotlin-analysis-compiler. Otherwise you may see obscure exceptions, such as NoSuchFieldError.

Executing the given example generates documentation in HTML output format.

See Command line options for more configuration details.

Run with JSON configuration

It's possible to configure the CLI runner with JSON. In this case, you need to provide an absolute/relative path to the JSON configuration file as the first and only argument. All other configuration options are parsed from it.

```
java -jar dokka-cli-1.8.20.jar dokka-configuration.json
```

At the very least, you need the following JSON configuration file:

```
{
  "outputDir": "./dokka/html",
  "sourceSets": [
    {
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "sourceRoots": [
        "/home/myCoolProject/src/main/kotlin"
      ]
    }
  ],
  "pluginsClasspath": [
    "./dokka-base-1.8.20.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./dokka-analysis-1.8.20.jar",
    "./kotlin-analysis-intellij-1.8.20.jar",
    "./kotlin-analysis-compiler-1.8.20.jar",
    "./freemarker-2.3.31.jar"
  ]
}
```

> Due to an internal class conflict, first pass kotlin-analysis-intellij and only then kotlin-analysis-compiler. Otherwise you may see obscure exceptions, such as NoSuchFieldError.

See JSON configuration options for more details.

Other output formats

By default, the dokka-base artifact contains the HTML output format only.

All other output formats are implemented as Dokka plugins. In order to use them, you have to put them on the plugins classpath.

For example, if you want to generate documentation in the experimental GFM output format, you need to download and pass gfm-plugin's JAR into the pluginsClasspath configuration option.

Via command line options:

```
java -jar dokka-cli-1.8.20.jar \
     -pluginsClasspath "./dokka-base-1.8.20.jar;...;./gfm-plugin-1.8.20.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.8.20.jar",
    "...",
    "./gfm-plugin-1.8.20.jar"
  ],
  ...
}
```

With the GFM plugin passed to pluginsClasspath, the CLI runner generates documentation in the GFM output format.

For more information, see Markdown and Javadoc pages.

Command line options

To see the list of all possible command line options and their detailed description, run:

```
java -jar dokka-cli-1.8.20.jar -help
```

Short summary:

Option	Description
moduleName	Name of the project/module.
moduleVersion	Documented version.
outputDir	Output directory path, ./dokka by default.
sourceSet	Configuration for a Dokka source set. Contains nested configuration options.
pluginsConfiguration	Configuration for Dokka plugins.
pluginsClasspath	List of jars with Dokka plugins and their dependencies. Accepts multiple paths separated by semicolons.
offlineMode	Whether to resolve remote files/links over network.
failOnWarning	Whether to fail documentation generation if Dokka has emitted a warning or an error.
delayTemplateSubstitution	Whether to delay substitution of some elements. Used in incremental builds of multi-module projects.
noSuppressObviousFunctions	Whether to suppress obvious functions such as those inherited from kotlin.Any and java.lang.Object.
includes	Markdown files that contain module and package documentation. Accepts multiple values separated by semicolons.
suppressInheritedMembers	Whether to suppress inherited members that aren't explicitly overridden in a given class.
globalPackageOptions	Global list of package configuration options in format "matchingRegex,-deprecated,-privateApi,+warnUndocumented,+suppress;+visibility:PUBLIC;...". Accepts multiple values separated by semicolons.
globalLinks	Global external documentation links in format {url}^{packageListUrl}. Accepts multiple values separated by ^^.
globalSrcLink	Global mapping between a source directory and a Web service for browsing the code. Accepts multiple paths separated by semicolons.
helpSourceSet	Prints help for the nested -sourceSet configuration.
loggingLevel	Logging level, possible values: DEBUG, PROGRESS, INFO, WARN, ERROR.

Option	Description
help, h	Usage info.

Source set options

To see the list of command line options for the nested -sourceSet configuration, run:

```
java -jar dokka-cli-1.8.20.jar -sourceSet -help
```

Short summary:

Option	Description
sourceSetName	Name of the source set.
displayName	Display name of the source set, used both internally and externally.
classpath	Classpath for analysis and interactive samples. Accepts multiple paths separated by semicolons.
src	Source code roots to be analyzed and documented. Accepts multiple paths separated by semicolons.
dependentSourceSets	Names of the dependent source sets in format moduleName/sourceSetName. Accepts multiple values separated by semicolons.
samples	List of directories or files that contain sample functions. Accepts multiple paths separated by semicolons.
includes	Markdown files that contain module and package documentation. Accepts multiple paths separated by semicolons.
documentedVisibilities	Visibilities to be documented. Accepts multiple values separated by semicolons. Possible values: PUBLIC, PRIVATE, PROTECTED, INTERNAL, PACKAGE.
reportUndocumented	Whether to report undocumented declarations.
noSkipEmptyPackages	Whether to create pages for empty packages.
skipDeprecated	Whether to skip deprecated declarations.
jdkVersion	Version of JDK to use for linking to JDK Javadocs.
languageVersion	Language version used for setting up analysis and samples.
apiVersion	Kotlin API version used for setting up analysis and samples.

Option	Description
noStdlibLink	Whether to generate links to the Kotlin standard library.
noJdkLink	Whether to generate links to JDK Javadocs.
suppressedFiles	Paths to files to be suppressed. Accepts multiple paths separated by semicolons.
analysisPlatform	Platform used for setting up analysis.
perPackageOptions	List of package source set configurations in format matchingRegexp,-deprecated,-privateApi,+warnUndocumented,+suppress;.... Accepts multiple values separated by semicolons.
externalDocumentationLinks	External documentation links in format {url}^{packageListUrl}. Accepts multiple values separated by ^^.
srcLink	Mapping between a source directory and a Web service for browsing the code. Accepts multiple paths separated by semicolons.

JSON configuration

Below are some examples and detailed descriptions for each configuration section. You can also find an example with all configuration options applied at the bottom of the page.

General configuration

```
{
  "moduleName": "Dokka Example",
  "moduleVersion": null,
  "outputDir": "./build/dokka/html",
  "failOnWarning": false,
  "suppressObviousFunctions": true,
  "suppressInheritedMembers": false,
  "offlineMode": false,
  "includes": [
    "module.md"
  ],
  "sourceLinks":  [
    { "_comment": "Options are described in a separate section" }
  ],
  "perPackageOptions": [
    { "_comment": "Options are described in a separate section" }
  ],
  "externalDocumentationLinks":  [
    { "_comment": "Options are described in a separate section" }
  ],
  "sourceSets": [
    { "_comment": "Options are described in a separate section" }
  ],
  "pluginsClasspath": [
    "./dokka-base-1.8.20.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./dokka-analysis-1.8.20.jar",
    "./kotlin-analysis-intellij-1.8.20.jar",
    "./kotlin-analysis-compiler-1.8.20.jar",
    "./freemarker-2.3.31.jar"
  ]
}
```

moduleName

The display name used to refer to the module. It is used for the table of contents, navigation, logging, etc.

Default: root

moduleVersion

The module version.

Default: empty

outputDirectory

The directory to where documentation is generated, regardless of output format.

Default: ./dokka

failOnWarning

Whether to fail documentation generation if Dokka has emitted a warning or an error. The process waits until all errors and warnings have been emitted first.

This setting works well with reportUndocumented

Default: false

suppressObviousFunctions

Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from kotlin.Any, Kotlin.Enum, java.lang.Object or java.lang.Enum, such as equals, hashCode, toString.
- Synthetic (generated by the compiler) and does not have any documentation, such as dataClass.componentN or dataClass.copy.

Default: true

suppressInheritedMembers

Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This can suppress functions such as equals / hashCode / toString, but cannot suppress synthetic functions such as dataClass.componentN and dataClass.copy. Use suppressObviousFunctions for that.

Default: false

offlineMode

Whether to resolve remote files/links over your network.

This includes package-lists used for generating external documentation links. For example, to make classes from the standard library clickable.

Setting this to true can significantly speed up build times in certain cases, but can also worsen documentation quality and user experience. For example, by not resolving class/member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See externalDocumentationLinks section.

Default: false

includes

A list of Markdown files that contain module and package documentation.

The contents of specified files are parsed and embedded into documentation as module and package descriptions.

This can be configured on per-package basis.

sourceSets

Individual and additional configuration of Kotlin source sets.

For a list of possible options, see source set configuration.

sourceLinks

The global configuration of source links that is applied for all source sets.

For a list of possible options, see source link configuration.

perPackageOptions

The global configuration of matched packages, regardless of the source set they are in.

For a list of possible options, see per-package configuration.

externalDocumentationLinks

The global configuration of external documentation links, regardless of the source set they are used in.

For a list of possible options, see external documentation links configuration.

pluginsClasspath

A list of JAR files with Dokka plugins and their dependencies.

Source set configuration

How to configure Kotlin source sets:

```
{
  "sourceSets": [
    {
      "displayName": "jvm",
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "dependentSourceSets": [
        {
          "scopeId": "dependentSourceSetScopeId",
          "sourceSetName": "dependentSourceSetName"
        }
      ],
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"],
      "reportUndocumented": false,
      "skipEmptyPackages": true,
      "skipDeprecated": false,
      "jdkVersion": 8,
      "languageVersion": "1.7",
      "apiVersion": "1.7",
      "noStdlibLink": false,
      "noJdkLink": false,
      "includes": [
        "module.md"
      ],
      "analysisPlatform": "jvm",
      "sourceRoots": [
        "/home/ignat/IdeaProjects/dokka-debug-mvn/src/main/kotlin"
      ],
      "classpath": [
        "libs/kotlin-stdlib-1.9.0.jar",
        "libs/kotlin-stdlib-common-1.9.0.jar"
      ],
      "samples": [
        "samples/basic.kt"
      ],
      "suppressedFiles": [
        "src/main/kotlin/org/jetbrains/dokka/Suppressed.kt"
      ],
      "sourceLinks":  [
        { "_comment": "Options are described in a separate section" }
      ],
      "perPackageOptions": [
        { "_comment": "Options are described in a separate section" }
      ],
      "externalDocumentationLinks":  [
        { "_comment": "Options are described in a separate section" }
      ]
    }
  ]
}
```

displayName

The display name used to refer to this source set.

The name is used both externally (for example, the source set name is visible to documentation readers) and internally (for example, for logging messages of reportUndocumented).

The platform name can be used if you don't have a better alternative.

sourceSetID

The technical ID of the source set

documentedVisibilities

The set of visibility modifiers that should be documented.

This can be used if you want to document protected/internal/private declarations, as well as if you want to exclude public declarations and only document internal API.

This can be configured on per-package basis.

Possible values:

- PUBLIC
- PRIVATE
- PROTECTED
- INTERNAL
- PACKAGE

Default: PUBLIC

reportUndocumented

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by documentedVisibilities and other filters.

This setting works well with failOnWarning.

This can be configured on per-package basis.

Default: false

skipEmptyPackages

Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if skipDeprecated is set to true and your package contains only deprecated declarations, it is considered to be empty.

Default for CLI runner is false.

skipDeprecated

Whether to document declarations annotated with @Deprecated.

This can be configured on per-package basis.

Default: false

jdkVersion

The JDK version to use when generating external documentation links for Java types.

For example, if you use java.util.UUID in some public declaration signature, and this option is set to 8, Dokka generates an external documentation link to JDK 8 Javadocs for it.

languageVersion

The Kotlin language version used for setting up analysis and @sample environment.

apiVersion

The Kotlin API version used for setting up analysis and @sample environment.

noStdlibLink

Whether to generate external documentation links that lead to the API reference documentation of Kotlin's standard library.

Note: Links are generated when noStdLibLink is set to false.

Default: false

noJdkLink

Whether to generate external documentation links to JDK's Javadocs.

The version of JDK Javadocs is determined by the jdkVersion option.

Note: Links are generated when noJdkLink is set to false.

Default: false

includes

A list of Markdown files that contain module and package documentation.

The contents of the specified files are parsed and embedded into documentation as module and package descriptions.

analysisPlatform

Platform to be used for setting up code analysis and @sample environment.

Possible values:

- jvm
- common
- js
- native

sourceRoots

The source code roots to be analyzed and documented. Acceptable inputs are directories and individual .kt / .java files.

classpath

The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved/picked up automatically.

This option accepts both .jar and .klib files.

samples

A list of directories or files that contain sample functions which are referenced via the @sample KDoc tag.

suppressedFiles

The files to be suppressed when generating documentation.

sourceLinks

A set of parameters for source links that is applied only for this source set.

For a list of possible options, see source link configuration.

perPackageOptions

A set of parameters specific to matched packages within this source set.

For a list of possible options, see per-package configuration.

externalDocumentationLinks

A set of parameters for external documentation links that is applied only for this source set.

For a list of possible options, see external documentation links configuration.

Source link configuration

The sourceLinks configuration block allows you to add a source link to each signature that leads to the remoteUrl with a specific line number. (The line number is configurable by setting remoteLineSuffix).

This helps readers to find the source code for each declaration.

For an example, see the documentation for the count() function in kotlinx.coroutines.

You can configure source links for all source sets together at the same time, or individually:

```
{
  "sourceLinks": [
    {
      "localDirectory": "src/main/kotlin",
      "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
      "remoteLineSuffix": "#L"
    }
  ]
}
```

localDirectory

The path to the local source directory.

remoteUrl

The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, etc. This URL is used to generate source code links of declarations.

remoteLineSuffix

The suffix used to append the source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to #L and the line number is 10, the resulting URL suffix is #L10.

Suffixes used by popular services:

- GitHub: #L
- GitLab: #L
- Bitbucket: #lines-

Default: empty (no suffix)

Per-package configuration

The perPackageOptions configuration block allows setting some options for specific packages matched by matchingRegex.

You can add package configurations for all source sets together at the same time, or individually:

```
{
  "perPackageOptions": [
    {
      "matchingRegex": ".*internal.*",
      "suppress": false,
      "skipDeprecated": false,
      "reportUndocumented": false,
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"]
    }
  ]
}
```

matchingRegex
The regular expression that is used to match the package.

suppress
Whether this package should be skipped when generating documentation.

Default: false

skipDeprecated
Whether to document declarations annotated with @Deprecated.

This can be set on project/module level.

Default: false

reportUndocumented
Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by documentedVisibilities and other filters.

This setting works well with failOnWarning.

This can be configured on source set level.

Default: false

documentedVisibilities
The set of visibility modifiers that should be documented.

This can be used if you want to document protected/internal/private declarations within this package, as well as if you want to exclude public declarations and only document internal API.

Can be configured on source set level.

Default: PUBLIC

External documentation links configuration

The externalDocumentationLinks block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from kotlinx.serialization, by default they are unclickable in your documentation, as if they are unresolved. However, since the API reference documentation for kotlinx.serialization is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. Thus allowing Dokka to generate links for types from the library, making them resolve successfully and clickable.

You can configure external documentation links for all source sets together at the same time, or individually:

```json
{
  "externalDocumentationLinks": [
    {
      "url": "https://kotlinlang.org/api/kotlinx.serialization/",
      "packageListUrl": "https://kotlinlang.org/api/kotlinx.serialization/package-list"
    }
  ]
}
```

url

The root URL of documentation to link to. It must contain a trailing slash.

Dokka does its best to automatically find package-list for the given URL, and link declarations together.

If automatic resolution fails or if you want to use locally cached files instead, consider setting the packageListUrl option.

packageListUrl

The exact location of a package-list. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as module and package names.

This can also be a locally cached file to avoid network calls.

Complete configuration

Below you can see all possible configuration options applied at the same time.

```json
{
  "moduleName": "Dokka Example",
  "moduleVersion": null,
  "outputDir": "./build/dokka/html",
  "failOnWarning": false,
  "suppressObviousFunctions": true,
  "suppressInheritedMembers": false,
  "offlineMode": false,
  "sourceLinks": [
    {
      "localDirectory": "src/main/kotlin",
      "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
      "remoteLineSuffix": "#L"
    }
  ],
  "externalDocumentationLinks": [
    {
      "url": "https://docs.oracle.com/javase/8/docs/api/",
      "packageListUrl": "https://docs.oracle.com/javase/8/docs/api/package-list"
    },
    {
      "url": "https://kotlinlang.org/api/latest/jvm/stdlib/",
      "packageListUrl": "https://kotlinlang.org/api/latest/jvm/stdlib/package-list"
    }
  ],
  "perPackageOptions": [
    {
      "matchingRegex": ".*internal.*",
      "suppress": false,
      "reportUndocumented": false,
      "skipDeprecated": false,
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"]
    }
  ],
  "sourceSets": [
    {
      "displayName": "jvm",
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "dependentSourceSets": [
        {
          "scopeId": "dependentSourceSetScopeId",
          "sourceSetName": "dependentSourceSetName"
        }
      ],
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"],
      "reportUndocumented": false,
      "skipEmptyPackages": true,
```

```
      "skipDeprecated": false,
      "jdkVersion": 8,
      "languageVersion": "1.7",
      "apiVersion": "1.7",
      "noStdlibLink": false,
      "noJdkLink": false,
      "includes": [
        "module.md"
      ],
      "analysisPlatform": "jvm",
      "sourceRoots": [
        "/home/ignat/IdeaProjects/dokka-debug-mvn/src/main/kotlin"
      ],
      "classpath": [
        "libs/kotlin-stdlib-1.9.0.jar",
        "libs/kotlin-stdlib-common-1.9.0.jar"
      ],
      "samples": [
        "samples/basic.kt"
      ],
      "suppressedFiles": [
        "src/main/kotlin/org/jetbrains/dokka/Suppressed.kt"
      ],
      "sourceLinks": [
        {
          "localDirectory": "src/main/kotlin",
          "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
          "remoteLineSuffix": "#L"
        }
      ],
      "externalDocumentationLinks": [
        {
          "url": "https://docs.oracle.com/javase/8/docs/api/",
          "packageListUrl": "https://docs.oracle.com/javase/8/docs/api/package-list"
        },
        {
          "url": "https://kotlinlang.org/api/latest/jvm/stdlib/",
          "packageListUrl": "https://kotlinlang.org/api/latest/jvm/stdlib/package-list"
        }
      ],
      "perPackageOptions": [
        {
          "matchingRegex": ".*internal.*",
          "suppress": false,
          "reportUndocumented": false,
          "skipDeprecated": false,
          "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"]
        }
      ]
    }
  ],
  "pluginsClasspath": [
    "./dokka-base-1.8.20.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./dokka-analysis-1.8.20.jar",
    "./kotlin-analysis-intellij-1.8.20.jar",
    "./kotlin-analysis-compiler-1.8.20.jar",
    "./freemarker-2.3.31.jar"
  ],
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"separateInheritedMembers\":false,\"footerMessage\":\"© 2021 pretty good Copyright\"}"
    }
  ],
  "includes": [
    "module.md"
  ]
}
```

HTML

HTML is Dokka's default and recommended output format. It is currently in Beta and approaching the Stable release.

You can see an example of the output by browsing documentation for kotlinx.coroutines.

Generate HTML documentation

HTML as an output format is supported by all runners. To generate HTML documentation, follow these steps depending on your build tool or runner:

- For Gradle, run dokkaHtml or dokkaHtmlMultiModule tasks.
- For Maven, run the dokka:dokka goal.
- For CLI runner, run with HTML dependencies set.

> HTML pages generated by this format need to be hosted on a web server in order to render everything correctly.
>
> You can use any free static site hosting service, such as GitHub Pages.
>
> Locally, you can use the built-in IntelliJ web server.

Configuration

HTML format is Dokka's base format, so it is configurable through DokkaBase and DokkaBaseConfiguration classes:

Kotlin

Via type-safe Kotlin DSL:

```
import org.jetbrains.dokka.base.DokkaBase
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.base.DokkaBaseConfiguration

buildscript {
    dependencies {
        classpath("org.jetbrains.dokka:dokka-base:1.8.20")
    }
}

tasks.withType<DokkaTask>().configureEach {
    pluginConfiguration<DokkaBase, DokkaBaseConfiguration> {
        customAssets = listOf(file("my-image.png"))
        customStyleSheets = listOf(file("my-styles.css"))
        footerMessage = "(c) 2022 MyOrg"
        separateInheritedMembers = false
        templatesDir = file("dokka/templates")
        mergeImplicitExpectActualDeclarations = false
    }
}
```

Via JSON:

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType<DokkaTask>().configureEach {
    val dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg",
      "separateInheritedMembers": false,
      "templatesDir": "${file("dokka/templates")}",
      "mergeImplicitExpectActualDeclarations": false
    }
    """
    pluginsMapConfiguration.set(
        mapOf(
            // fully qualified plugin name to json configuration
            "org.jetbrains.dokka.base.DokkaBase" to dokkaBaseConfiguration
        )
    )
}
```

Groovy

```groovy
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType(DokkaTask.class) {
    String dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
      "separateInheritedMembers": false,
      "templatesDir": "${file("dokka/templates")}",
      "mergeImplicitExpectActualDeclarations": false
    }
    """
    pluginsMapConfiguration.set(
            // fully qualified plugin name to json configuration
            ["org.jetbrains.dokka.base.DokkaBase": dokkaBaseConfiguration]
    )
}
```

Maven

```xml
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <pluginsConfiguration>
            <!-- Fully qualified plugin name -->
            <org.jetbrains.dokka.base.DokkaBase>
                <!-- Options by name -->
                <customAssets>
                    <asset>${project.basedir}/my-image.png</asset>
                </customAssets>
                <customStyleSheets>
                    <stylesheet>${project.basedir}/my-styles.css</stylesheet>
                </customStyleSheets>
                <footerMessage>(c) MyOrg 2022 Maven</footerMessage>
                <separateInheritedMembers>false</separateInheritedMembers>
                <templatesDir>${project.basedir}/dokka/templates</templatesDir>
                <mergeImplicitExpectActualDeclarations>false</mergeImplicitExpectActualDeclarations>
            </org.jetbrains.dokka.base.DokkaBase>
        </pluginsConfiguration>
    </configuration>
</plugin>
```

CLI

Via command line options:

```bash
java -jar dokka-cli-1.8.20.jar \
     ...
     -pluginsConfiguration "org.jetbrains.dokka.base.DokkaBase={\"customAssets\": [\"my-image.png\"], \"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\": \"(c) 2022 MyOrg\", \"separateInheritedMembers\": false, \"templatesDir\": \"dokka/templates\", \"mergeImplicitExpectActualDeclarations\": false}
"
```

Via JSON configuration:

```json
{
  "moduleName": "Dokka Example",
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"customAssets\": [\"my-image.png\"], \"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\": \"(c) 2022 MyOrg\", \"separateInheritedMembers\": false, \"templatesDir\": \"dokka/templates\", \"mergeImplicitExpectActualDeclarations\": false}"
    }
  ]
}
```

Configuration options

The table below contains all of the possible configuration options and their purpose.

Option	Description
customAssets	List of paths for image assets to be bundled with documentation. The image assets can have any file extension. For more information, see Customizing assets.
customStyleSheets	List of paths for .css stylesheets to be bundled with documentation and used for rendering. For more information, see Customizing styles.
templatesDir	Path to the directory containing custom HTML templates. For more information, see Templates.
footerMessage	The text displayed in the footer.
separateInheritedMembers	This is a boolean option. If set to true, Dokka renders properties/functions and inherited properties/inherited functions separately. This is disabled by default.
mergeImplicitExpectActualDeclarations	This is a boolean option. If set to true, Dokka merges declarations that are not declared as expect/actual, but have the same fully qualified name. This can be useful for legacy codebases. This is disabled by default.

For more information about configuring Dokka plugins, see Configuring Dokka plugins.

Customization

To help you add your own look and feel to your documentation, the HTML format supports a number of customization options.

Customize styles

You can use your own stylesheets by using the customStyleSheets configuration option. These are applied to every page.

It's also possible to override Dokka's default stylesheets by providing files with the same name:

Stylesheet name	Description
style.css	Main stylesheet, contains most of the styles used across all pages
logo-styles.css	Header logo styling
prism.css	Styles for PrismJS syntax highlighter

The source code for all of Dokka's stylesheets is available on GitHub.

Customize assets

You can provide your own images to be bundled with documentation by using the customAssets configuration option.

These files are copied to the <output>/images directory.

It's possible to override Dokka's images and icons by providing files with the same name. The most useful and relevant one being logo-icon.svg, which is the image that's used in the header. The rest is mostly icons.

You can find all images used by Dokka on GitHub.

Change the logo

To customize the logo, you can begin by providing your own asset for logo-icon.svg.

If you don't like how it looks, or you want to use a .png file instead of the default .svg file, you can override the logo-styles.css stylesheet to customize it.

For an example of how to do this, see our custom format example project.

Modify the footer

You can modify text in the footer by using the footerMessage configuration option.

Templates

Dokka provides the ability to modify FreeMarker templates used for generating documentation pages.

You can change the header completely, add your own banners/menus/search, load analytics, change body styling and so on.

Dokka uses the following templates:

Template	Description
base.ftl	Defines the general design of all pages to be rendered.
includes/header.ftl	The page header that by default contains the logo, version, source set selector, light/dark theme switch, and search.
includes/footer.ftl	The page footer that contains the footerMessage configuration option and copyright.
includes/page_metadata.ftl	Metadata used within <head> container.
includes/source_set_selector.ftl	The source set selector in the header.

The base template is base.ftl and it includes all of the remaining listed templates. You can find the source code for all of Dokka's templates on GitHub.

You can override any template by using the templatesDir configuration option. Dokka searches for the exact template names within the given directory. If it fails to find user-defined templates, it uses the default templates.

Variables

The following variables are available inside all templates:

Variable	Description
${pageName}	The page name
${footerMessage}	The text which is set by the footerMessage configuration option
${sourceSets}	A nullable list of source sets for multi-platform pages. Each item has name, platform, and filter properties.
${projectName}	The project name. It's available only within the template_cmd directive.
${pathToRoot}	The path to root from the current page. It's useful for locating assets and is available only within the template_cmd directive.

Variables projectName and pathToRoot are available only within the template_cmd directive as they require more context and thus they need to be resolved at later stages by the MultiModule task:

```
<@template_cmd name="projectName">
   <span>${projectName}</span>
</@template_cmd>
```

Directives

You can also use the following Dokka-defined directives:

Variable	Description
<@content/>	The main page content.
<@resources/>	Resources such as scripts and stylesheets.
<@version/>	The module version taken from configuration. If the versioning plugin is applied, it is replaced with a version navigator.

Markdown

> The Markdown output formats are still in Alpha, so you may find bugs and experience migration issues when using them. You use them at your own risk.

Dokka is able to generate documentation in GitHub Flavored and Jekyll compatible Markdown.

These formats give you more freedom in terms of hosting documentation as the output can be embedded right into your documentation website. For example, see OkHttp's API reference pages.

Markdown output formats are implemented as Dokka plugins, maintained by the Dokka team, and they are open source.

GFM

The GFM output format generates documentation in GitHub Flavored Markdown.

Gradle

The Gradle plugin for Dokka comes with the GFM output format included. You can use the following tasks with it:

Task	Description
dokkaGfm	Generates GFM documentation for a single project.
dokkaGfmMultiModule	A MultiModule task created only for parent projects in multi-project builds. It generates documentation for subprojects and collects all outputs in a single place with a common table of contents.
dokkaGfmCollector	A Collector task created only for parent projects in multi-project builds. It callsdokkaGfm for every subproject and merges all outputs into a single virtual project.

Maven

Since GFM format is implemented as a Dokka plugin, you need to apply it as a plugin dependency:

```xml
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>gfm-plugin</artifactId>
                <version>1.8.20</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

After configuring this, running the dokka:dokka goal produces documentation in GFM format.

For more information, see the Mavin plugin documentation for Other output formats.

CLI

Since GFM format is implemented as a Dokka plugin, you need to download the JAR file and pass it to pluginsClasspath.

Via command line options:

```
java -jar dokka-cli-1.8.20.jar \
     -pluginsClasspath "./dokka-base-1.8.20.jar;...;./gfm-plugin-1.8.20.jar" \
     ...
```

Via JSON configuration:

```json
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.8.20.jar",
    "...",
    "./gfm-plugin-1.8.20.jar"
  ],
  ...
}
```

For more information, see the CLI runner's documentation for Other output formats.

You can find the source code on GitHub.

Jekyll

The Jekyll output format generates documentation in Jekyll compatible Markdown.

Gradle

The Gradle plugin for Dokka comes with the Jekyll output format included. You can use the following tasks with it:

Task	Description
dokkaJekyll	Generates Jekyll documentation for a single project.
dokkaJekyllMultiModule	A MultiModule task created only for parent projects in multi-project builds. It generates documentation for subprojects and collects all outputs in a single place with a common table of contents.
dokkaJekyllCollector	A Collector task created only for parent projects in multi-project builds. It calls dokkaJekyll for every subproject and merges all outputs into a single virtual project.

Maven

Since Jekyll format is implemented as a Dokka plugin, you need to apply it as a plugin dependency:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>jekyll-plugin</artifactId>
                <version>1.8.20</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

After configuring this, running the dokka:dokka goal produces documentation in GFM format.

For more information, see the Maven plugin's documentation for Other output formats.

CLI

Since Jekyll format is implemented as a Dokka plugin, you need to download the JAR file. This format is also based on GFM format, so you need to provide it as a dependency as well. Both JARs need to be passed to pluginsClasspath:

Via command line options:

```
java -jar dokka-cli-1.8.20.jar \
     -pluginsClasspath "./dokka-base-1.8.20.jar;...;./gfm-plugin-1.8.20.jar;./jekyll-plugin-1.8.20.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.8.20.jar",
    "...",
    "./gfm-plugin-1.8.20.jar",
    "./jekyll-plugin-1.8.20.jar"
  ],
  ...
}
```

For more information, see the CLI runner's documentation for Other output formats.

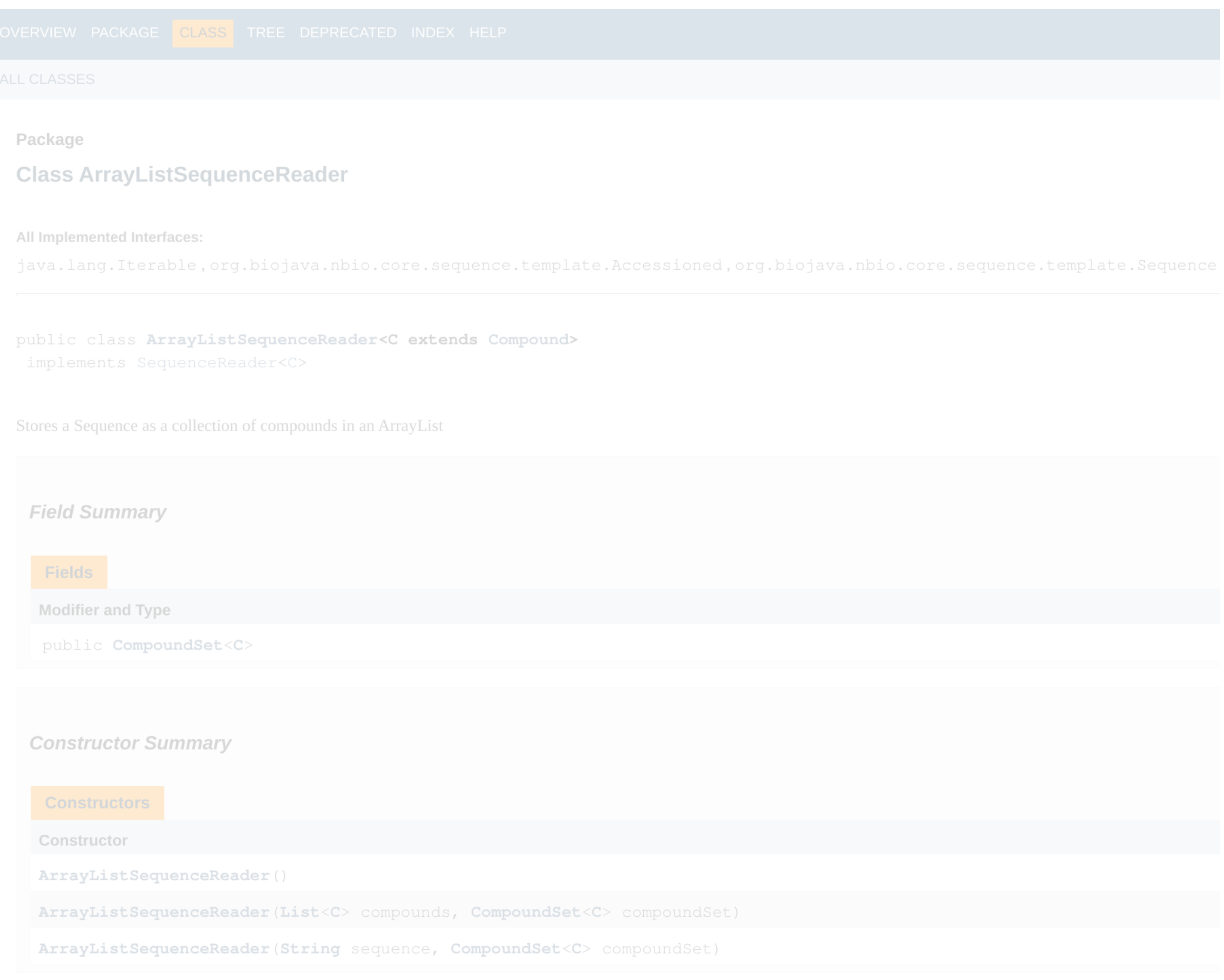
You can find the source code on GitHub.

Javadoc

> The Javadoc output format is still in Alpha, so you may find bugs and experience migration issues when using it. Successful integration with tools that accept Java's Javadoc HTML as input is not guaranteed. You use it at your own risk.

Dokka's Javadoc output format is a lookalike of Java's Javadoc HTML format.

It tries to visually mimic HTML pages generated by the Javadoc tool, but it's not a direct implementation or an exact copy.



Screenshot of javadoc output format

All Kotlin code and signatures are rendered as seen from Java's perspective. This is achieved with our Kotlin as Java Dokka plugin, which comes bundled and applied by default for this format.

The Javadoc output format is implemented as a Dokka plugin, and it is maintained by the Dokka team. It is open source and you can find the source code on GitHub.

Generate Javadoc documentation

> The Javadoc format does not support multiplatform projects.

Gradle

The Gradle plugin for Dokka comes with the Javadoc output format included. You can use the following tasks:

Task	Description
dokkaJavadoc	Generates Javadoc documentation for a single project.

Task	Description
dokkaJavadocCollector	A Collector task created only for parent projects in multi-project builds. It callsdokkaJavadoc for every subproject and merges all outputs into a single virtual project.

The javadoc.jar file can be generated separately. For more information, seeBuilding javadoc.jar.

Maven

The Maven plugin for Dokka comes with the Javadoc output format built in. You can generate documentation by using the following goals:

Goal	Description
dokka:javadoc	Generates documentation in Javadoc format
dokka:javadocJar	Generates a javadoc.jar file that contains documentation in Javadoc format

CLI

Since the Javadoc output format is a Dokka plugin, you need to download the plugin'sJAR file.

The Javadoc output format has two dependencies that you need to provide as additional JAR files:

- kotlin-as-java plugin
- korte-jvm

Via command line options:

```
java -jar dokka-cli-1.8.20.jar \
     -pluginsClasspath "./dokka-base-1.8.20.jar;...;./javadoc-plugin-1.8.20.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.8.20.jar",
    "...",
    "./kotlin-as-java-plugin-1.8.20.jar",
    "./korte-jvm-3.3.0.jar",
    "./javadoc-plugin-1.8.20.jar"
  ],
  ...
}
```

For more information, see Other output formats in the CLI runner's documentation.

Dokka plugins

Dokka was built from the ground up to be easily extensible and highly customizable, which allows the community to implement plugins for missing or very specific features that are not provided out of the box.

Dokka plugins range anywhere from supporting other programming language sources to exotic output formats. You can add support for your own KDoc tags or annotations, teach Dokka how to render different DSLs that are found in KDoc descriptions, visually redesign Dokka's pages to be seamlessly integrated into your company's website, integrate it with other tools and so much more.

If you want to learn how to create Dokka plugins, see Developer guides.

Apply Dokka plugins

Dokka plugins are published as separate artifacts, so to apply a Dokka plugin you only need to add it as a dependency. From there, the plugin extends Dokka by itself - no further action is needed.

> Plugins that use the same extension points or work in a similar way can interfere with each other. This may lead to visual bugs, general undefined behaviour or even failed builds. However, it should not lead to concurrency issues since Dokka does not expose any mutable data structures or objects.
>
> If you notice problems like this, it's a good idea to check which plugins are applied and what they do.

Let's have a look at how you can apply the mathjax plugin to your project:

Kotlin

The Gradle plugin for Dokka creates convenient dependency configurations that allow you to apply plugins universally or for a specific output format only.

```
dependencies {
    // Is applied universally
    dokkaPlugin("org.jetbrains.dokka:mathjax-plugin:1.8.20")

    // Is applied for the single-module dokkaHtml task only
    dokkaHtmlPlugin("org.jetbrains.dokka:kotlin-as-java-plugin:1.8.20")

    // Is applied for HTML format in multi-project builds
    dokkaHtmlPartialPlugin("org.jetbrains.dokka:kotlin-as-java-plugin:1.8.20")
}
```

> When documenting multi-project builds, you need to apply Dokka plugins within subprojects as well as in their parent project.

Groovy

The Gradle plugin for Dokka creates convenient dependency configurations that allow you to apply Dokka plugins universally or for a specific output format only.

```
dependencies {
    // Is applied universally
    dokkaPlugin 'org.jetbrains.dokka:mathjax-plugin:1.8.20'

    // Is applied for the single-module dokkaHtml task only
    dokkaHtmlPlugin 'org.jetbrains.dokka:kotlin-as-java-plugin:1.8.20'

    // Is applied for HTML format in multi-project builds
    dokkaHtmlPartialPlugin 'org.jetbrains.dokka:kotlin-as-java-plugin:1.8.20'
}
```

> When documenting multi-project builds, you need to apply Dokka plugins within subprojects as well as in their parent project.

Maven

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>mathjax-plugin</artifactId>
                <version>1.8.20</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

CLI

If you are using the CLI runner with command line options, Dokka plugins should be passed as .jar files to -pluginsClasspath:

```
java -jar dokka-cli-1.8.20.jar \
     -pluginsClasspath "./dokka-base-1.8.20.jar;...;./mathjax-plugin-1.8.20.jar" \
     ...
```

If you are using JSON configuration, Dokka plugins should be specified under pluginsClasspath.

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.8.20.jar",
    "...",
    "./mathjax-plugin-1.8.20.jar"
  ],
  ...
}
```

Configure Dokka plugins

Dokka plugins can also have configuration options of their own. To see which options are available, consult the documentation of the plugins you are using.

Let's have a look at how you can configure the DokkaBase plugin, which is responsible for generating HTML documentation, by adding a custom image to the assets (customAssets option), by adding custom style sheets (customStyleSheets option), and by modifying the footer message (footerMessage option):

Kotlin

Gradle's Kotlin DSL allows for type-safe plugin configuration. This is achievable by adding the plugin's artifact to the classpath dependencies in the buildscript block, and then importing plugin and configuration classes:

```
import org.jetbrains.dokka.base.DokkaBase
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.base.DokkaBaseConfiguration

buildscript {
    dependencies {
        classpath("org.jetbrains.dokka:dokka-base:1.8.20")
    }
}

tasks.withType<DokkaTask>().configureEach {
    pluginConfiguration<DokkaBase, DokkaBaseConfiguration> {
        customAssets = listOf(file("my-image.png"))
        customStyleSheets = listOf(file("my-styles.css"))
        footerMessage = "(c) 2022 MyOrg"
    }
}
```

Alternatively, plugins can be configured via JSON. With this method, no additional dependencies are needed.

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType<DokkaTask>().configureEach {
    val dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
    }
    """
    pluginsMapConfiguration.set(
        mapOf(
            // fully qualified plugin name to json configuration
            "org.jetbrains.dokka.base.DokkaBase" to dokkaBaseConfiguration
        )
    )
}
```

Groovy

```groovy
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType(DokkaTask.class) {
    String dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
    }
    """
    pluginsMapConfiguration.set(
            // fully qualified plugin name to json configuration
            ["org.jetbrains.dokka.base.DokkaBase": dokkaBaseConfiguration]
    )
}
```

Maven

```xml
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <pluginsConfiguration>
            <!-- Fully qualified plugin name -->
            <org.jetbrains.dokka.base.DokkaBase>
                <!-- Options by name -->
                <customAssets>
                    <asset>${project.basedir}/my-image.png</asset>
                </customAssets>
                <customStyleSheets>
                    <stylesheet>${project.basedir}/my-styles.css</stylesheet>
                </customStyleSheets>
                <footerMessage>(c) MyOrg 2022 Maven</footerMessage>
            </org.jetbrains.dokka.base.DokkaBase>
        </pluginsConfiguration>
    </configuration>
</plugin>
```

CLI

If you are using the CLI runner with command line options, use the -pluginsConfiguration option that accepts JSON configuration in the form offullyQualifiedPluginName=json.

If you need to configure multiple plugins, you can pass multiple values separated by^^.

```bash
java -jar dokka-cli-1.8.20.jar \
     ...
     -pluginsConfiguration "org.jetbrains.dokka.base.DokkaBase={\"customAssets\": [\"my-image.png\"], \"customStyleSheets\": [\"my-
styles.css\"], \"footerMessage\": \"(c) 2022 MyOrg CLI\"}"
```

If you are using JSON configuration, there exists a similar pluginsConfiguration array that accepts JSON configuration in values.

```json
{
  "moduleName": "Dokka Example",
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"customAssets\": [\"my-image.png\"], \"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\": \"(c) 2022
MyOrg\"}"
    }
  ]
}
```

Notable plugins

Here are some notable Dokka plugins that you might find useful:

Name	Description

Name	Description
Android documentation plugin	Improves the documentation experience on Android
Versioning plugin	Adds version selector and helps to organize documentation for different versions of your application/library
MermaidJS HTML plugin	Renders MermaidJS diagrams and visualizations found in KDocs
Mathjax HTML plugin	Pretty prints mathematics found in KDocs
Kotlin as Java plugin	Renders Kotlin signatures as seen from Java's perspective

If you are a Dokka plugin author and would like to add your plugin to this list, get in touch with maintainers via Slack or GitHub.

Module documentation

Documentation for a module as a whole, as well as packages in that module, can be provided as separate Markdown files.

File format

Inside the Markdown file, the documentation for the module as a whole and for individual packages is introduced by the corresponding first-level headings. The text of the heading must be Module <module name> for a module, and Package <package qualified name> for a package.

The file doesn't have to contain both module and package documentation. You can have files that contain only package or module documentation. You can even have a Markdown file per module or package.

Using Markdown syntax, you can add:

- Headings up to level 6
- Emphasis with bold or italic formatting
- Links
- Inline code
- Code blocks
- Blockquotes

Here's an example file containing both module and package documentation:

```
# Module kotlin-demo

This content appears under your module name.

# Package org.jetbrains.kotlin.demo

This content appears under your package name in the packages list.
It also appears under the first-level heading on your package's page.

## Level 2 heading for package org.jetbrains.kotlin.demo

Content after this heading is also part of documentation for `org.jetbrains.kotlin.demo`

# Package org.jetbrains.kotlin.demo2

This content appears under your package name in the packages list.
```

```
It also appears under the first-level heading on your package's page.

## Level 2 heading for package org.jetbrains.kotlin.demo2

Content after this heading is also part of documentation for `org.jetbrains.kotlin.demo2`
```

To explore an example project with Gradle, see Dokka gradle example.

Pass files to Dokka

To pass these files to Dokka, you need to use the relevant includes option for Gradle, Maven, or CLI:

Gradle

Use the includes option in Source set configuration.

Maven

Use the includes option in General configuration.

CLI

If you are using command line configuration, use the includes option in Source set options.

If you are using JSON configuration, use the includes option in General configuration.

IDEs for Kotlin development

JetBrains provides the official Kotlin plugin for two Integrated Development Environments (IDEs): IntelliJ IDEA and Android Studio.

Other IDEs and source editors, such as Eclipse, Visual Studio Code, and Atom, have Kotlin community-supported plugins.

IntelliJ IDEA

IntelliJ IDEA is an IDE for JVM languages designed to maximize developer productivity. It does the routine and repetitive tasks for you by providing clever code completion, static code analysis, and refactorings, and lets you focus on the bright side of software development, making it not only productive but also an enjoyable experience.

Kotlin plugin is bundled with each IntelliJ IDEA release.

Read more about IntelliJ IDEA in the official documentation.

Android Studio

Android Studio is the official IDE for Android app development, based on IntelliJ IDEA. On top of IntelliJ's powerful code editor and developer tools, Android Studio offers even more features that enhance your productivity when building Android apps.

Kotlin plugin is bundled with each Android Studio release.

Read more about Android Studio in the official documentation.

Eclipse

Eclipse is an IDE that is used to develop applications in different programming languages, including Kotlin. Eclipse also has the Kotlin plugin: originally developed by JetBrains, now the Kotlin plugin is supported by the Kotlin community contributors.

You can install the Kotlin plugin manually from the Eclipse Marketplace.

The Kotlin team manages the development and contribution process to the Kotlin plugin for Eclipse. If you want to contribute to the plugin, send a pull request to the Kotlin for Eclipse repository on GitHub.

Compatibility with the Kotlin language versions

For IntelliJ IDEA and Android Studio the Kotlin plugin is bundled with each IDE release. When the new Kotlin version is released, these IDEs will suggest updating Kotlin to the latest version automatically. See the latest supported language version for each IDE in Kotlin releases.

Other IDEs support

JetBrains doesn't provide the Kotlin plugin for other IDEs. However, some of the other IDEs and source editors, such as Eclipse, Visual Studio Code, and Atom, have their own Kotlin plugins supported by the Kotlin community.

You can use any text editor to write the Kotlin code, but without IDE-related features: code formatting, debugging tools, and so on. To use Kotlin in text editors, you can download the latest Kotlin command-line compiler (kotlin-compiler-1.9.0.zip) from Kotlin GitHub Releases and install it manually. Also, you could use package managers, such as Homebrew, SDKMAN!, and Snap package.

What's next?

- Start your first project using IntelliJ IDEA IDE
- Create your first cross-platform mobile app using Android Studio
- Learn how to install EAP version of the Kotlin plugin

Migrate to Kotlin code style

Kotlin coding conventions and IntelliJ IDEA formatter

Kotlin coding conventions affect several aspects of writing idiomatic Kotlin, and a set of formatting recommendations aimed at improving Kotlin code readability is among them.

Unfortunately, the code formatter built into IntelliJ IDEA had to work long before this document was released and now has a default setup that produces different formatting from what is now recommended.

It may seem a logical next step to remove this obscurity by switching the defaults in IntelliJ IDEA and make formatting consistent with the Kotlin coding conventions. But this would mean that all the existing Kotlin projects will have a new code style enabled the moment the Kotlin plugin is installed. Not really the expected result for plugin update, right?

That's why we have the following migration plan instead:

- Enable the official code style formatting by default starting from Kotlin 1.3 and only for new projects (old formatting can be enabled manually)
- Authors of existing projects may choose to migrate to the Kotlin coding conventions
- Authors of existing projects may choose to explicitly declare using the old code style in a project (this way the project won't be affected by switching to the defaults in the future)
- Switch to the default formatting and make it consistent with Kotlin coding conventions in Kotlin 1.4

Differences between "Kotlin coding conventions" and "IntelliJ IDEA default code style"

The most notable change is in the continuation indentation policy. There's a nice idea to use the double indent for showing that a multi-line expression hasn't ended on the previous line. This is a very simple and general rule, but several Kotlin constructions look a bit awkward when they are formatted this way. In Kotlin coding conventions, it's recommended to use a single indent in cases where the long continuation indent has been forced before.

```kotlin
class User(
        val name: String,
        val address: String) {
    val key: String = getUserId(name, address)
}

fun findUser(
        users: List<User>,
        key: String? = null,
        name: String? = null): User? {
    val filteredUsers = users
            .asSequence()
            .filter { user -> key == null || user.key == key }
            .filter { user -> name == null || user.name.contains(name) }

    return filteredUsers.single()
}
```

```kotlin
class User(
    val name: String,
    val address: String
) {
    val key: String = getUserId(name, address)
}

fun findUser(
    users: List<User>,
    key: String? = null,
    name: String? = null
): User? {
    val filteredUsers = users
        .asSequence()
        .filter { user -> key == null || user.key == key }
        .filter { user -> name == null || user.name.contains(name) }

    return filteredUsers.single()
}
```

Code formatting

In practice, quite a bit of code is affected, so this can be considered a major code style update.

Migration to a new code style discussion

A new code style adoption might be a very natural process if it starts with a new project, when there's no code formatted in the old way. That is why starting from version 1.3, the Kotlin IntelliJ Plugin creates new projects with formatting from the Coding conventions document which is enabled by default.

Changing formatting in an existing project is a far more demanding task, and should probably be started with discussing all the caveats with the team.

The main disadvantage of changing the code style in an existing project is that the blame/annotate VCS feature will point to irrelevant commits more often. While each VCS has some kind of way to deal with this problem ("Annotate Previous Revision" can be used in IntelliJ IDEA), it's important to decide if a new style is worth all the effort. The practice of separating reformatting commits from meaningful changes can help a lot with later investigations.

Also migrating can be harder for larger teams because committing a lot of files in several subsystems may produce merging conflicts in personal branches. And while each conflict resolution is usually trivial, it's still wise to know if there are large feature branches currently in work.

In general, for small projects, we recommend converting all the files at once.

For medium and large projects the decision may be tough. If you are not ready to update many files right away you may decide to migrate module by module, or continue with gradual migration for modified files only.

Migration to a new code style

Switching to the Kotlin Coding Conventions code style can be done in Settings/Preferences | Editor | Code Style | Kotlin dialog. Switch scheme to Project and activate Set from... | Kotlin style guide.

In order to share those changes for all project developers .idea/codeStyle folder have to be committed to VCS.

If an external build system is used for configuring the project, and it's been decided not to share .idea/codeStyle folder, Kotlin coding conventions can be forced with an additional property:

In Gradle

Add kotlin.code.style=official property to the gradle.properties file at the project root and commit the file to VCS.

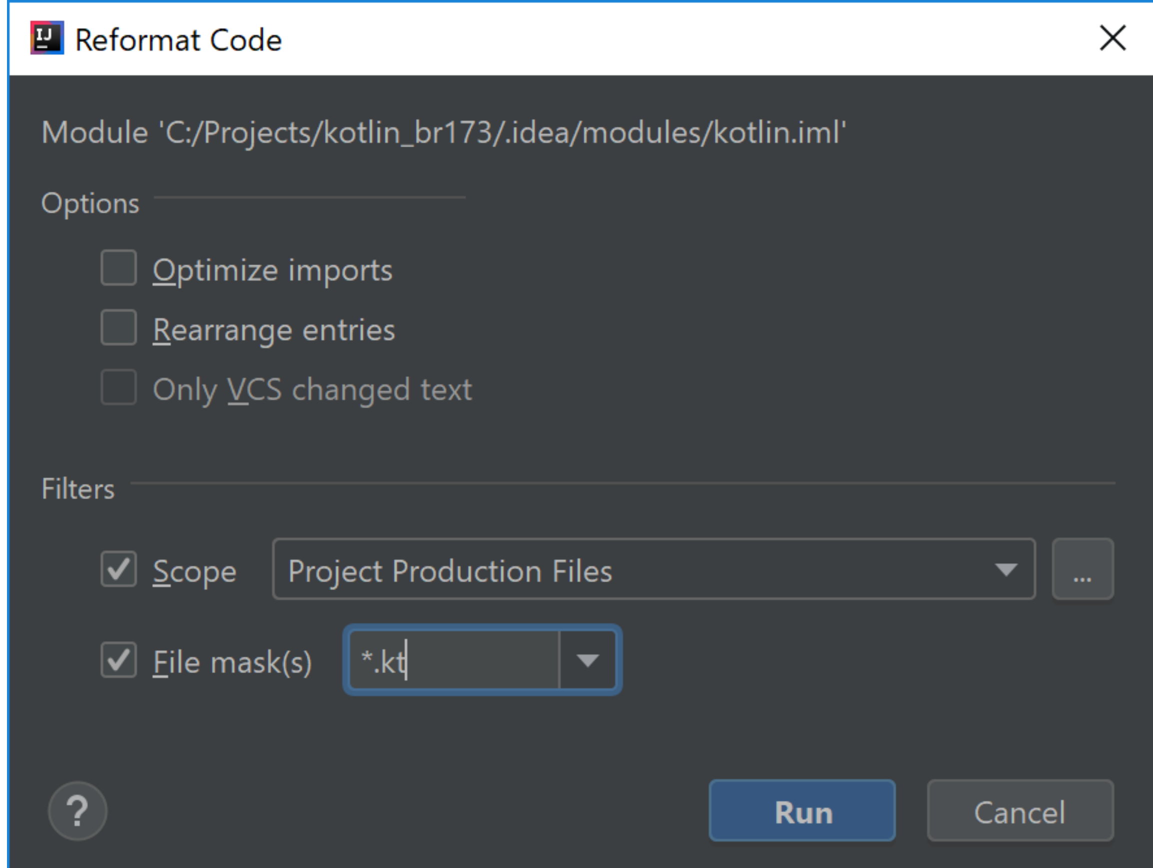
In Maven

Add kotlin.code.style official property to root pom.xml project file.

<properties> <kotlin.code.style>official</kotlin.code.style> </properties>

> Having the kotlin.code.style option set may modify the code style scheme during a project import and may change the code style settings.

After updating your code style settings, activate Reformat Code in the project view on the desired scope.



Reformat code

For a gradual migration, it's possible to enable the File is not formatted according to project settings inspection. It will highlight the places that should be reformatted. After enabling the Apply only to modified files option, inspection will show formatting problems only in modified files. Such files are probably going to be committed soon anyway.

Store old code style in project

It's always possible to explicitly set the IntelliJ IDEA code style as the correct code style for the project:

1. In Settings/Preferences | Editor | Code Style | Kotlin, switch to the Project scheme.
2. Open the Load/Save tab and in the Use defaults from select Kotlin obsolete IntelliJ IDEA codestyle.

In order to share the changes across the project developers .idea/codeStyle folder, it has to be committed to VCS. Alternatively, kotlin.code.style=obsolete can be used for projects configured with Gradle or Maven.

Run code snippets

Kotlin code is typically organized into projects with which you work in an IDE, a text editor, or another tool. However, if you want to quickly see how a function works or find an expression's value, there's no need to create a new project and build it. Check out these three handy ways to run Kotlin code instantly in different environments:

- Scratch files and worksheets in the IDE.
- Kotlin Playground in the browser.
- ki shell in the command line.

IDE: scratches and worksheets

IntelliJ IDEA and Android Studio support Kotlin scratch files and worksheets.

- Scratch files (or just scratches) let you create code drafts in the same IDE window as your project and run them on the fly. Scratches are not tied to projects; you can access and run all your scratches from any IntelliJ IDEA window on your OS.

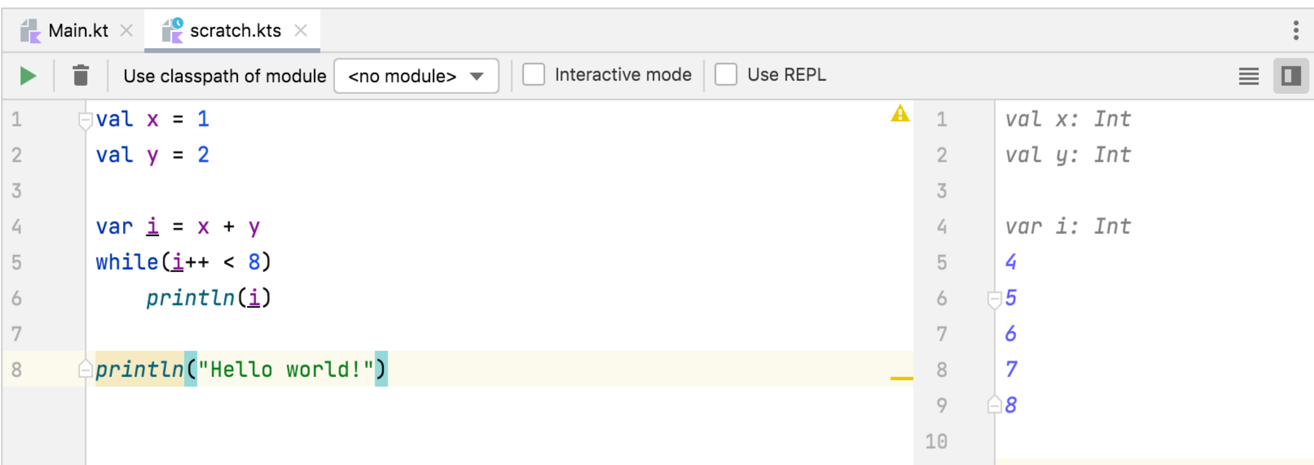
  To create a Kotlin scratch, click File | New | Scratch File and select the Kotlin type.

- Worksheets are project files: they are stored in project directories and tied to the project modules. Worksheets are useful for writing pieces of code that don't actually make a software unit but should still be stored together in a project, such as educational or demo materials.

  To create a Kotlin worksheet in a project directory, right-click the directory in the project tree and select New | Kotlin Worksheet.

Syntax highlighting, auto-completion, and other IntelliJ IDEA code editing features are supported in scratches and worksheets. There's no need to declare the main() function – all the code you write is executed as if it were in the body of main().

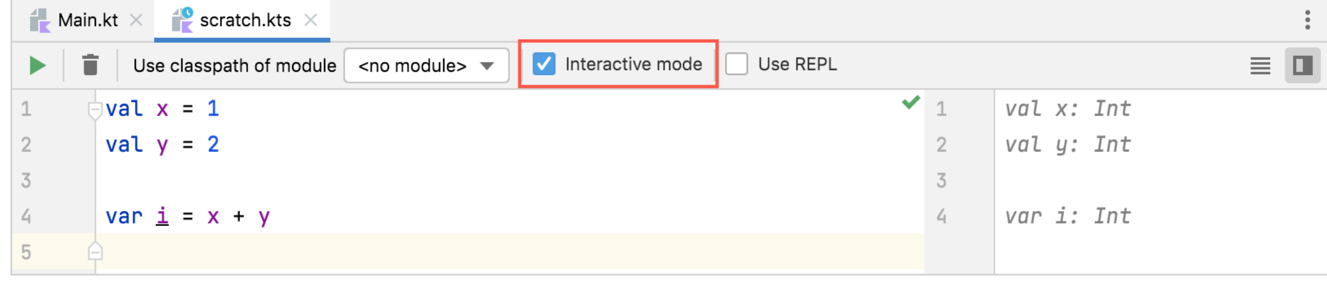
Once you have finished writing your code in a scratch or a worksheet, click Run. The execution results will appear in the lines opposite your code.



Run scratch

Interactive mode

The IDE can run code from scratches and worksheets automatically. To get execution results as soon as you stop typing, switch on Interactive mode.



Scratch interactive mode

Use modules

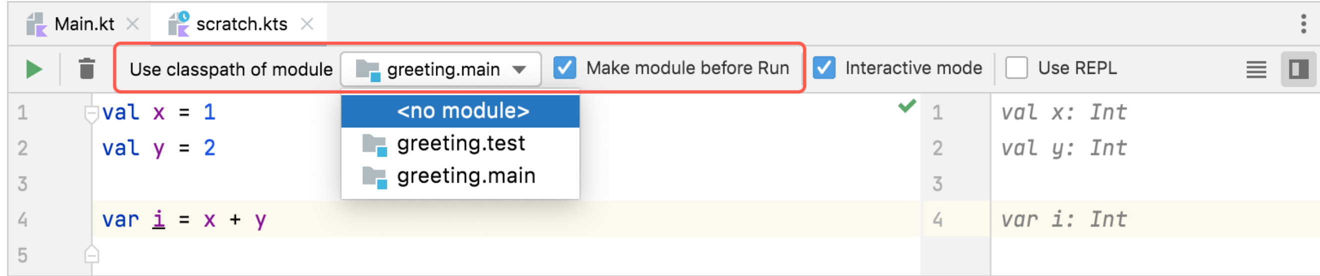
You can use classes or functions from a Kotlin project in your scratches and worksheets.

Worksheets automatically have access to classes and functions from the module where they reside.

To use classes or functions from a project in a scratch, import them into the scratch file with the import statement, as usual. Then write your code and run it with the appropriate module selected in the Use classpath of module list.

Both scratches and worksheets use the compiled versions of connected modules. So, if you modify a module's source files, the changes will propagate to scratches and worksheets when you rebuild the module. To rebuild the module automatically before each run of a scratch or a worksheet, select Make module

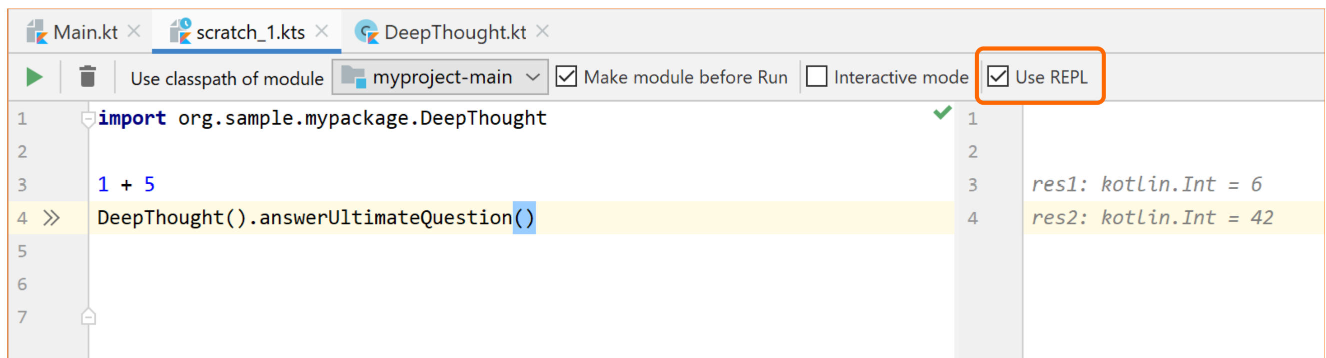
before Run.



Scratch select module

Run as REPL

To evaluate each particular expression in a scratch or a worksheet, run it with Use REPL selected. The code lines will run sequentially, providing the results of each call. You can later use the results in the same file by reffering to their auto-generated res* names (they are shown in the corresponding lines).



Scratch REPL

Browser: Kotlin Playground

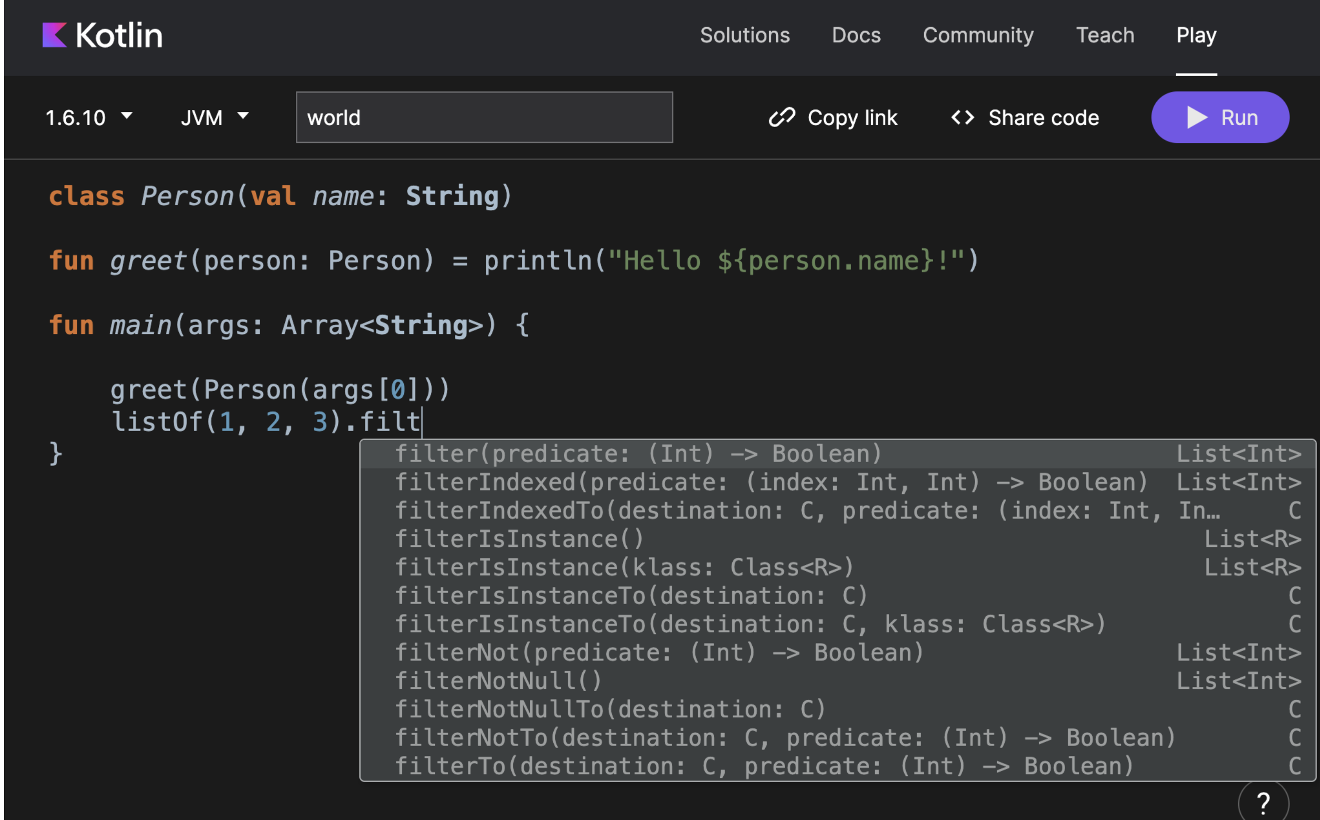
Kotlin Playground is an online application for writing, running, and sharing Kotlin code in your browser.

Write and edit code

In the Playground's editor area, you can write code just as you would in a source file:

- Add your own classes, functions, and top-level declarations in an arbitrary order.
- Write the executable part in the body of the main() function.

As in typical Kotlin projects, the main() function in the Playground can have the args parameter or no parameters at all. To pass program arguments upon execution, write them in the Program arguments field.



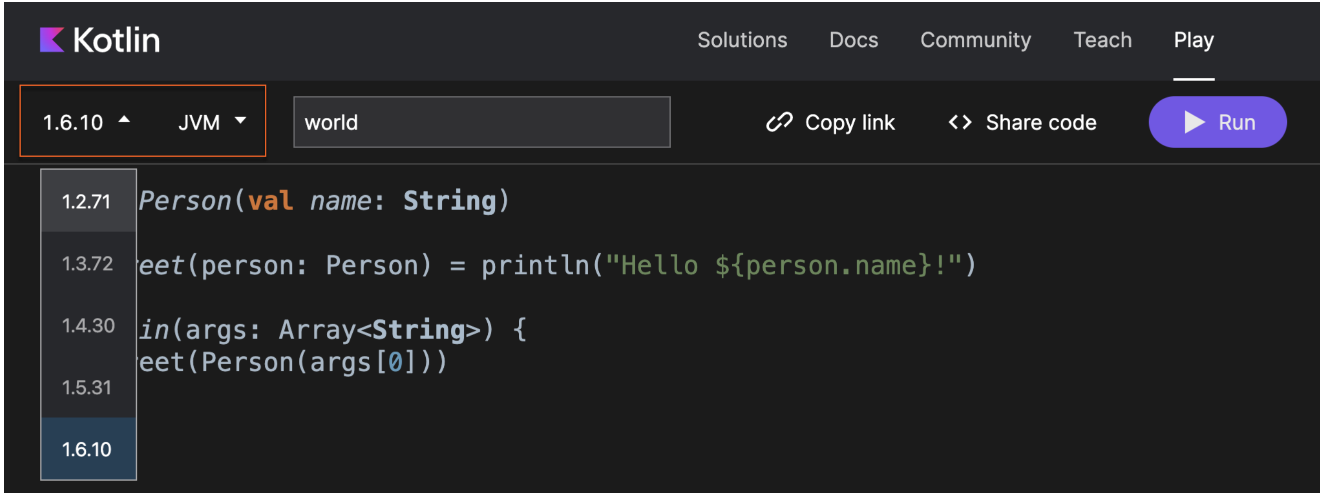
Playground: code completion

The Playground highlights the code and shows code completion options as you type. It automatically imports declarations from the standard library and kotlinx.coroutines.

Choose execution environment

The Playground provides ways to customize the execution environment:

- Multiple Kotlin versions, including available previews of future versions.
- Multiple backends to run the code in: JVM, JS (legacy or IR compiler, or Canvas), or JUnit.



Playground: environment setup

For JS backends, you can also see the generated JS code.

Kotlin Solutions Docs Community Teach Play

1.6.10 JS Generated JS Copy link Share code Run

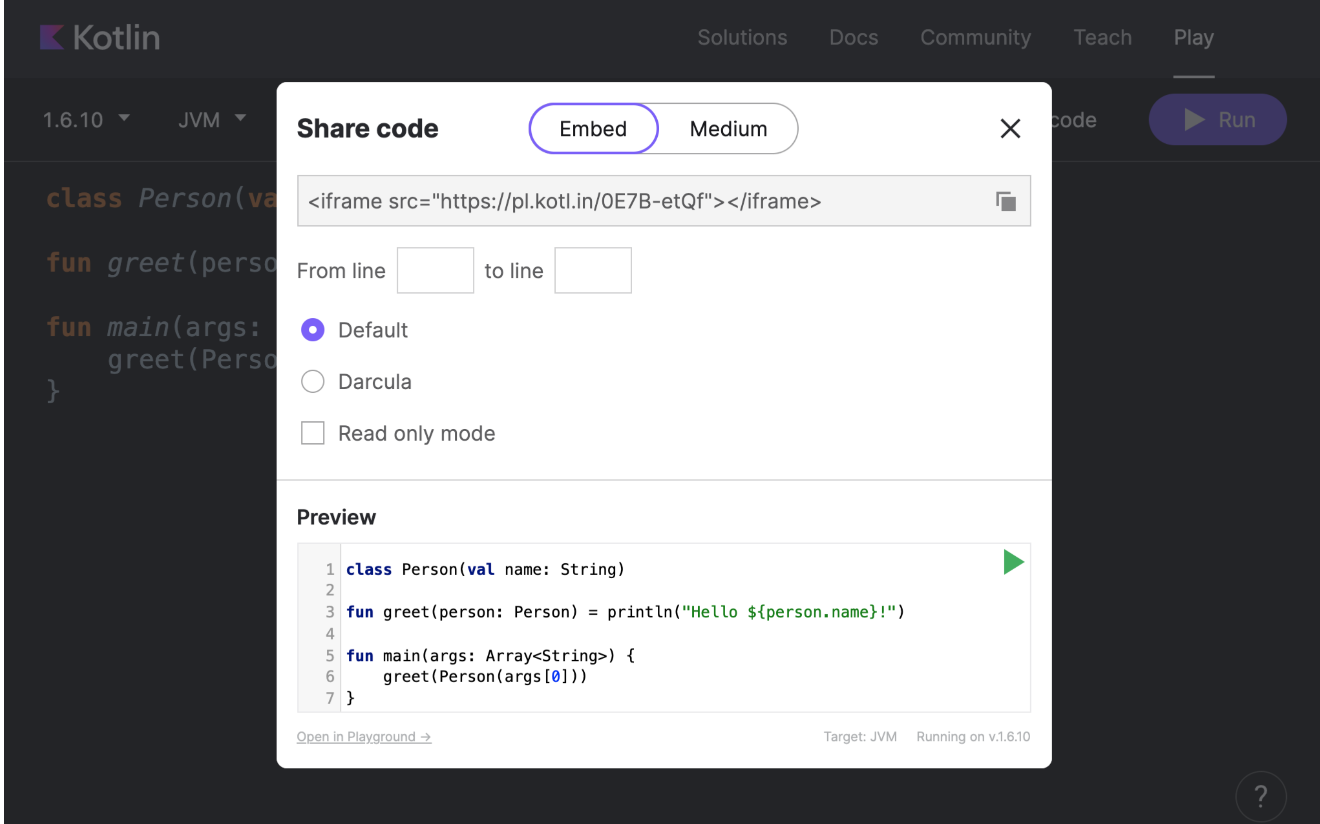
```
class Person(val name: String)

fun greet(person: Person) = println("Hello ${person.name}!")

fun main(args: Array<String>) {
    greet(Person(args[0]))
}
```

Playground: generated JS

Share code online

Use the Playground to share your code with others – click Copy link and send it to anyone you want to show the code to.

You can also embed code snippets from the Playground into other websites and even make them runnable. Click Share code to embed your sample into any web page or into a Medium article.



Playground: share code

Command line: ki shell

The ki shell (Kotlin Interactive Shell) is a command-line utility for running Kotlin code in the terminal. It's available for Linux, macOS, and Windows.

The ki shell provides basic code evaluation capabilities, along with advanced features such as:

* code completion
* type checks
* external dependencies
* paste mode for code snippets
* scripting support

See the ki shell GitHub repository for more details.

Install and run ki shell

To install the ki shell, download the latest version of it from GitHub and unzip it in the directory of your choice.

On macOS, you can also install the ki shell with Homebrew by running the following command:

```
brew install ki
```

To start the ki shell, run bin/ki.sh on Linux and macOS (or just ki if the ki shell was installed with Homebrew) or bin\ki.bat on Windows.

Once the shell is running, you can immediately start writing Kotlin code in your terminal. Type :help (or :h) to see the commands that are available in the ki shell.

Code completion and highlighting

The ki shell shows code completion options when you press Tab. It also provides syntax highlighting as you type. You can disable this feature by entering :syntax off.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] val x = 1 + 2 + 3
[1] var greeting = "Hello"
[2] listOf(1, 2, 3).fil
file                  filterIndexedTo(      filterIsInstanceTo(   filterNotNullTo(
filter {              filterIsInstance(     filterNot {           filterNotTo(
filterIndexed {       filterIsInstance()    filterNotNull()       filterTo(
```

ki shell highlighting and completion

When you press Enter, the ki shell evaluates the entered line and prints the result. Expression values are printed as variables with auto-generated names like res*. You can later use such variables in the code you run. If the construct entered is incomplete (for example, an if with a condition but without the body), the shell prints three dots and expects the remaining part.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] val greeting = "Hello"
[1] greeting + " world!"
res1: String = Hello world!
[2] println(res1)
Hello world!
[3] if (res1.length > 10)
...
```

ki shell results

Check an expression's type

For complex expressions or APIs that you don't know well, the ki shell provides the :type (or :t) command, which shows the type of an expression:

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :t sequenceOf("one", "two", "three").associateWith { it.length }
Map<String, Int>
[1]
```

ki shell type

Load code

If the code you need is stored somewhere else, there are two ways to load it and use it in the ki shell:

- Load a source file with the :load (or :l) command.
- Copy and paste the code snippet in paste mode with the :paste (or :p) command.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :l Desktop/sourceFile.kt
[1] :ls
val variableFromFile: String
fun functionFromFile(): Unit
[2] println(variableFromFile)
Hello from file!
[3] functionFromFile()
Calling function from file
[4]
```

ki shell load file

The ls command shows available symbols (variables and functions).

Add external dependencies

Along with the standard library, the ki shell also supports external dependencies. This lets you try out third-party libraries in it without creating a whole project.

To add a third-party library in the ki shell, use the :dependsOn command. By default, the ki shell works with Maven Central, but you can use other repositories if you connect them using the :repository command:

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :repository https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven
[1] :dependsOn org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3
[2] import kotlinx.html.*
[3] import kotlinx.html.stream.*
[4] val html = createHTML().html {
...body {
...h1 { "Hello" }
...}
...}
[9] html
res5: String = <html>
  <body>
    <h1></h1>
  </body>
</html>

[10]
```

ki shell external dependency

Kotlin and continuous integration with TeamCity

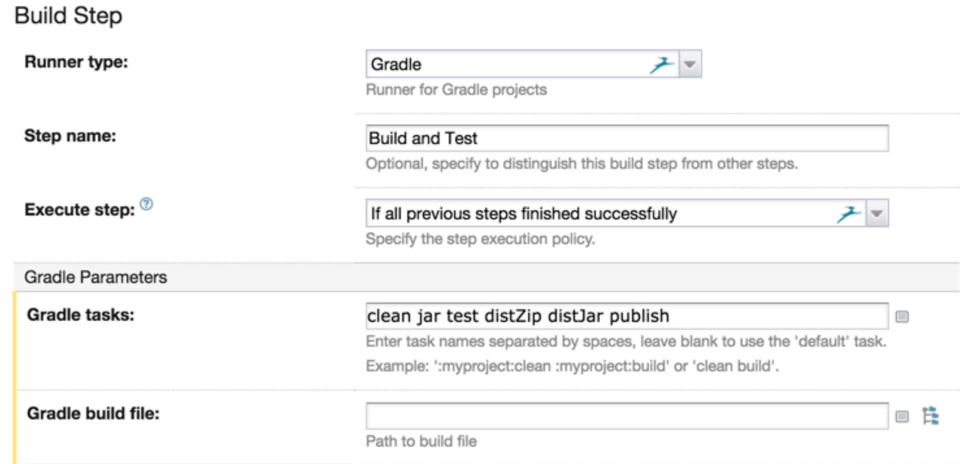
On this page, you'll learn how to set up TeamCity to build your Kotlin project. For more information and basics of TeamCity please check the Documentation page

which contains information about installation, basic configuration, etc.

Kotlin works with different build tools, so if you're using a standard tool such as Ant, Maven or Gradle, the process for setting up a Kotlin project is no different to any other language or library that integrates with these tools. Where there are some minor requirements and differences is when using the internal build system of IntelliJ IDEA, which is also supported on TeamCity.

Gradle, Maven, and Ant

If using Ant, Maven or Gradle, the setup process is straightforward. All that is needed is to define the Build Step. For example, if using Gradle, simply define the required parameters such as the Step Name and Gradle tasks that need executing for the Runner Type.



Gradle Build Step

Since all the dependencies required for Kotlin are defined in the Gradle file, nothing else needs to be configured specifically for Kotlin to run correctly.

If using Ant or Maven, the same configuration applies. The only difference being that the Runner Type would be Ant or Maven respectively.

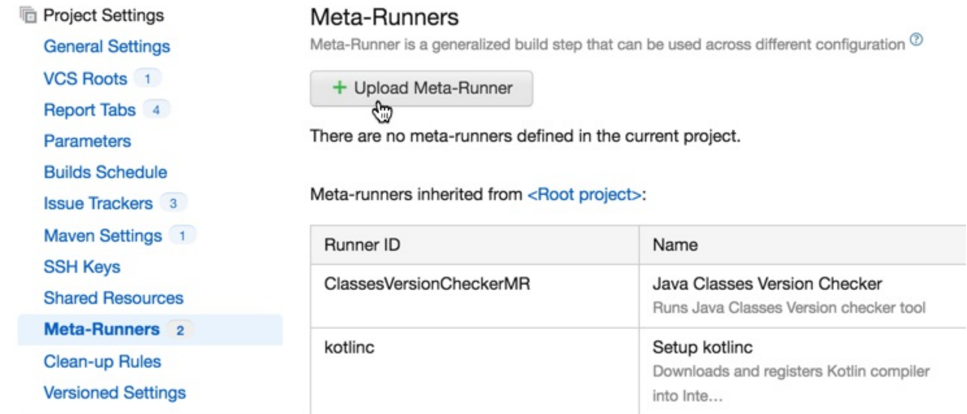
IntelliJ IDEA Build System

If using IntelliJ IDEA build system with TeamCity, make sure that the version of Kotlin being used by IntelliJ IDEA is the same as the one that TeamCity runs. You may need to download the specific version of the Kotlin plugin and install it on TeamCity.

Fortunately, there is a meta-runner already available that takes care of most of the manual work. If not familiar with the concept of TeamCity meta-runners, check the documentation. They are very easy and powerful way to introduce custom Runners without the need to write plugins.

Download and install the meta-runner

The meta-runner for Kotlin is available on GitHub. Download that meta-runner and import it from the TeamCity user interface



Meta-runner

Setup Kotlin compiler fetching step
Basically this step is limited to defining the Step Name and the version of Kotlin you need. Tags can be used.

New Build Step

Runner type: Setup kotlinc
Downloads and registers Kotlin compiler into IntelliJ IDEA build runner

Step name:
Optional, specify to distinguish this build step from other steps.

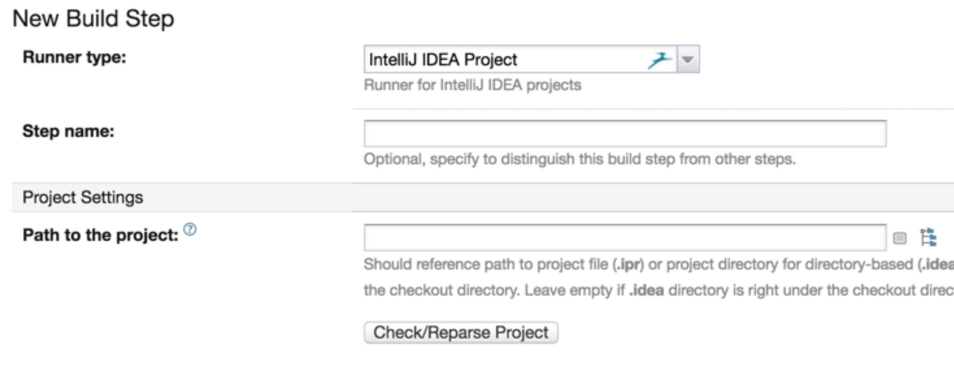
Execute step: If all previous steps finished successfully
Specify the step execution policy.

Kotlin Version M10
Select Kotlin build tag, e.g. 'M9 or bootstrap'

Setup Kotlin Compiler

The runner will set the value for the property system.path.macro.KOTLIN.BUNDLED to the correct one based on the path settings from the IntelliJ IDEA project. However, this value needs to be defined in TeamCity (and can be set to any value). Therefore, you need to define it as a system variable.

Setup Kotlin compilation step
The final step is to define the actual compilation of the project, which uses the standard IntelliJ IDEA Runner Type.



IntelliJ IDEA Runner

With that, our project should now build and produce the corresponding artifacts.

Other CI servers

If using a continuous integration tool different to TeamCity, as long as it supports any of the build tools, or calling command line tools, compiling Kotlin and automating things as part of a CI process should be possible.

Document Kotlin code: KDoc

The language used to document Kotlin code (the equivalent of Java's Javadoc) is called KDoc. In essence, KDoc combines Javadoc's syntax for block tags (extended to support Kotlin's specific constructs) and Markdown for inline markup.

> Kotlin's documentation engine: Dokka, understands KDoc and can be used to generate documentation in various formats. For more information, read our Dokka documentation.

KDoc syntax

Just like with Javadoc, KDoc comments start with /** and end with */. Every line of the comment may begin with an asterisk, which is not considered part of the contents of the comment.

By convention, the first paragraph of the documentation text (the block of text until the first blank line) is the summary description of the element, and the following text is the detailed description.

Every block tag begins on a new line and starts with the @ character.

Here's an example of a class documented using KDoc:

```
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```

Block tags

KDoc currently supports the following block tags:

@param name

Documents a value parameter of a function or a type parameter of a class, property or function. To better separate the parameter name from the description, if you prefer, you can enclose the name of the parameter in brackets. The following two syntaxes are therefore equivalent:

@param name description. @param[name] description.

@return

Documents the return value of a function.

@constructor

Documents the primary constructor of a class.

@receiver
Documents the receiver of an extension function.

@property name
Documents the property of a class which has the specified name. This tag can be used for documenting properties declared in the primary constructor, where putting a doc comment directly before the property definition would be awkward.

@throws class, @exception class
Documents an exception which can be thrown by a method. Since Kotlin does not have checked exceptions, there is also no expectation that all possible exceptions are documented, but you can still use this tag when it provides useful information for users of the class.

@sample identifier
Embeds the body of the function with the specified qualified name into the documentation for the current element, in order to show an example of how the element could be used.

@see identifier
Adds a link to the specified class or method to the See also block of the documentation.

@author
Specifies the author of the element being documented.

@since
Specifies the version of the software in which the element being documented was introduced.

@suppress
Excludes the element from the generated documentation. Can be used for elements which are not part of the official API of a module but still have to be visible externally.

> KDoc does not support the @deprecated tag. Instead, please use the @Deprecated annotation.

Inline markup

For inline markup, KDoc uses the regular Markdown syntax, extended to support a shorthand syntax for linking to other elements in the code.

Links to elements
To link to another element (class, method, property, or parameter), simply put its name in square brackets:

```
Use the method [foo] for this purpose.
```

If you want to specify a custom label for the link, add it in another set of square brackets before the element link:

```
Use [this method][foo] for this purpose.
```

You can also use qualified names in the element links. Note that, unlike Javadoc, qualified names always use the dot character to separate the components, even before a method name:

```
Use [kotlin.reflect.KClass.properties] to enumerate the properties of the class.
```

Names in element links are resolved using the same rules as if the name was used inside the element being documented. In particular, this means that if you have imported a name into the current file, you don't need to fully qualify it when you use it in a KDoc comment.

Note that KDoc does not have any syntax for resolving overloaded members in links. Since Kotlin's documentation generation tool puts the documentation for all overloads of a function on the same page, identifying a specific overloaded function is not required for the link to work.

External links

To add an external link, use the typical Markdown syntax:

```
For more information about KDoc syntax, see [KDoc](<example-URL>).
```

What's next?

Learn how to use Kotlin's documentation generation tool: Dokka.

Kotlin and OSGi

To enable Kotlin OSGi support in your Kotlin project, include kotlin-osgi-bundle instead of the regular Kotlin libraries. It is recommended to remove kotlin-runtime, kotlin-stdlib and kotlin-reflect dependencies as kotlin-osgi-bundle already contains all of them. You also should pay attention in case when external Kotlin libraries are included. Most regular Kotlin dependencies are not OSGi-ready, so you shouldn't use them and should remove them from your project.

Maven

To include the Kotlin OSGi bundle to a Maven project:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-osgi-bundle</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

To exclude the standard library from external libraries (notice that "star exclusion" works in Maven 3 only):

```
<dependency>
    <groupId>some.group.id</groupId>
    <artifactId>some.library</artifactId>
    <version>some.library.version</version>

    <exclusions>
        <exclusion>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Gradle

To include kotlin-osgi-bundle to a Gradle project:

Kotlin

```
dependencies {
    implementation(kotlin("osgi-bundle"))
}
```

Groovy

```
dependencies {
```

```
    implementation "org.jetbrains.kotlin:kotlin-osgi-bundle:1.9.0"
}
```

To exclude default Kotlin libraries that comes as transitive dependencies you can use the following approach:

Kotlin

```kotlin
dependencies {
    implementation("some.group.id:some.library:someversion") {
        exclude(group = "org.jetbrains.kotlin")
    }
}
```

Groovy

```groovy
dependencies {
    implementation('some.group.id:some.library:someversion') {
        exclude group: 'org.jetbrains.kotlin'
    }
}
```

FAQ

Why not just add required manifest options to all Kotlin libraries

Even though it is the most preferred way to provide OSGi support, unfortunately it couldn't be done for now due to so called "package split" issue that couldn't be easily eliminated and such a big change is not planned for now. There is Require-Bundle feature but it is not the best option too and not recommended to use. So it was decided to make a separate artifact for OSGi.

Kotlin command-line compiler

Every Kotlin release ships with a standalone version of the compiler. You can download the latest version manually or via a package manager.

> Installing the command-line compiler is not an essential step to use Kotlin. A general way to write Kotlin applications is using an IDE - IntelliJ IDEA or Android Studio. They provide full Kotlin support out of the box without needing additional components. Learn how to get started with Kotlin in an IDE.

Install the compiler

Manual install

1. Download the latest version (kotlin-compiler-1.9.0.zip) from GitHub Releases.
2. Unzip the standalone compiler into a directory and optionally add the bin directory to the system path. The bin directory contains the scripts needed to compile and run Kotlin on Windows, macOS, and Linux.

SDKMAN!

An easier way to install Kotlin on UNIX-based systems, such as macOS, Linux, Cygwin, FreeBSD, and Solaris, is SDKMAN!. It also works in Bash and ZSH shells. Learn how to install SDKMAN!.

To install the Kotlin compiler via SDKMAN!, run the following command in the terminal:

```
sdk install kotlin
```

Homebrew

Alternatively, on macOS you can install the compiler via Homebrew:

```
brew update
brew install kotlin
```

Snap package

If you use Snap on Ubuntu 16.04 or later, you can install the compiler from the command line:

```
sudo snap install --classic kotlin
```

Create and run an application

1. Create a simple application in Kotlin that displays "Hello, World!". In your favorite editor, create a new file called hello.kt with the following lines:

   ```
   fun main() {
       println("Hello, World!")
   }
   ```

2. Compile the application using the Kotlin compiler:

   ```
   kotlinc hello.kt -include-runtime -d hello.jar
   ```

   The -d option indicates the output path for generated class files, which may be either a directory or a .jar file. The -include-runtime option makes the resulting .jar file self-contained and runnable by including the Kotlin runtime library in it.

   To see all available options, run

   ```
   kotlinc -help
   ```

3. Run the application.

   ```
   java -jar hello.jar
   ```

Compile a library

If you're developing a library to be used by other Kotlin applications, you can build the .jar file without including the Kotlin runtime in it:

```
kotlinc hello.kt -d hello.jar
```

Since binaries compiled this way depend on the Kotlin runtime, you should make sure the latter is present in the classpath whenever your compiled library is used.

You can also use the kotlin script to run binaries produced by the Kotlin compiler:

```
kotlin -classpath hello.jar HelloKt
```

HelloKt is the main class name that the Kotlin compiler generates for the file named hello.kt.

Run the REPL

You can run the compiler without parameters to have an interactive shell. In this shell, you can type any valid Kotlin code and see the results.

```
[Ocean] ~/tutorials/kotlin/command_line/kotlinc$ bin/kotlinc-jvm
Kotlin interactive shell
Type :help for help, :quit for quit
>>> 2+2
4
>>> println("Welcome to the Kotlin Shell")
Welcome to the Kotlin Shell
>>>
```

Shell

Run scripts

Kotlin can also be used as a scripting language. A script is a Kotlin source file (.kts) with top-level executable code.

```
import java.io.File

// Get the passed in path, i.e. "-d some/path" or use the current path.
val path = if (args.contains("-d")) args[1 + args.indexOf("-d")]
           else "."

val folders = File(path).listFiles { file -> file.isDirectory() }
folders?.forEach { folder -> println(folder) }
```

To run a script, pass the -script option to the compiler with the corresponding script file:

```
kotlinc -script list_folders.kts -- -d <path_to_folder_to_inspect>
```

Kotlin provides experimental support for script customization, such as adding external properties, providing static or dynamic dependencies, and so on. Customizations are defined by so-called Script definitions - annotated kotlin classes with the appropriate support code. The script filename extension is used to select the appropriate definition. Learn more about Kotlin custom scripting.

Properly prepared script definitions are detected and applied automatically when the appropriate jars are included in the compilation classpath. Alternatively, you can specify definitions manually by passing the -script-templates option to the compiler:

```
kotlinc -script-templates org.example.CustomScriptDefinition -script custom.script1.kts
```

For additional details, please consult the KEEP-75.

Kotlin compiler options

Each release of Kotlin includes compilers for the supported targets: JVM, JavaScript, and native binaries for supported platforms.

These compilers are used by:

- The IDE, when you click the Compile or Run button for your Kotlin project.
- Gradle, when you call gradle build in a console or in the IDE.
- Maven, when you call mvn compile or mvn test-compile in a console or in the IDE.

You can also run Kotlin compilers manually from the command line as described in the Working with command-line compiler tutorial.

Compiler options

Kotlin compilers have a number of options for tailoring the compiling process. Compiler options for different targets are listed on this page together with a description of each one.

There are several ways to set the compiler options and their values (compiler arguments):

- In IntelliJ IDEA, write in the compiler arguments in the Additional command line parameters text box in Settings/Preferences | Build, Execution, Deployment | Compiler | Kotlin Compiler.

- If you're using Gradle, specify the compiler arguments in the compilerOptions property of the Kotlin compilation task. For details, see Gradle compiler options.
- If you're using Maven, specify the compiler arguments in the <configuration> element of the Maven plugin node. For details, see Maven.
- If you run a command-line compiler, add the compiler arguments directly to the utility call or write them into an argfile.

For example:

```
$ kotlinc hello.kt -include-runtime -d hello.jar
```

> On Windows, when you pass compiler arguments that contain delimiter characters (whitespace, =, ;, ,), surround these arguments with double quotes (").
>
> $ kotlinc.bat hello.kt -include-runtime -d "My Folder\hello.jar"

Common options

The following options are common for all Kotlin compilers.

-version
Display the compiler version.

-nowarn
Suppress the compiler from displaying warnings during compilation.

-Werror
Turn any warnings into a compilation error.

-verbose
Enable verbose logging output which includes details of the compilation process.

-script
Evaluate a Kotlin script file. When called with this option, the compiler executes the first Kotlin script (*.kts) file among the given arguments.

-help (-h)
Display usage information and exit. Only standard options are shown. To show advanced options, use -X.

-X
Display information about the advanced options and exit. These options are currently unstable: their names and behavior may be changed without notice.

-kotlin-home path
Specify a custom path to the Kotlin compiler used for the discovery of runtime libraries.

-P plugin:pluginId:optionName=value
Pass an option to a Kotlin compiler plugin. Available plugins and their options are listed in the Tools > Compiler plugins section of the documentation.

-language-version version
Provide source compatibility with the specified version of Kotlin.

-api-version version
Allow using declarations only from the specified version of Kotlin bundled libraries.

-progressive
Enable the progressive mode for the compiler.

In the progressive mode, deprecations and bug fixes for unstable code take effect immediately, instead of going through a graceful migration cycle. Code written in the progressive mode is backwards compatible; however, code written in a non-progressive mode may cause compilation errors in the progressive mode.

@argfile
Read the compiler options from the given file. Such a file can contain compiler options with values and paths to the source files. Options and paths should be separated by whitespaces. For example:

-include-runtime -d hello.jar hello.kt

To pass values that contain whitespaces, surround them with single (') or double (") quotes. If a value contains quotation marks in it, escape them with a backslash (\).

-include-runtime -d 'My folder'

You can also pass multiple argument files, for example, to separate compiler options from source files.

```
$ kotlinc @compiler.options @classes
```

If the files reside in locations different from the current directory, use relative paths.

```
$ kotlinc @options/compiler.options hello.kt
```

-opt-in annotation
Enable usages of API that requires opt-in with a requirement annotation with the given fully qualified name.

Kotlin/JVM compiler options

The Kotlin compiler for JVM compiles Kotlin source files into Java class files. The command-line tools for Kotlin to JVM compilation are kotlinc and kotlinc-jvm. You can also use them for executing Kotlin script files.

In addition to the common options, Kotlin/JVM compiler has the options listed below.

-classpath path (-cp path)
Search for class files in the specified paths. Separate elements of the classpath with system path separators (; on Windows, : on macOS/Linux). The classpath can contain file and directory paths, ZIP, or JAR files.

-d path
Place the generated class files into the specified location. The location can be a directory, a ZIP, or a JAR file.

-include-runtime
Include the Kotlin runtime into the resulting JAR file. Makes the resulting archive runnable on any Java-enabled environment.

-jdk-home path
Use a custom JDK home directory to include into the classpath if it differs from the default JAVA_HOME.

-Xjdk-release=version
Specify the target version of the generated JVM bytecode. Limit the API of the JDK in the classpath to the specified Java version. Automatically sets -jvm-target

version. Possible values are 1.8, 9, 10, ..., 20. The default value is 1.8.

> This option is not guaranteed to be effective for each JDK distribution.

-jvm-target version

Specify the target version of the generated JVM bytecode. Possible values are 1.8, 9, 10, ..., 20. The default value is 1.8.

-java-parameters

Generate metadata for Java 1.8 reflection on method parameters.

-module-name name (JVM)

Set a custom name for the generated .kotlin_module file.

-no-jdk

Don't automatically include the Java runtime into the classpath.

-no-reflect

Don't automatically include the Kotlin reflection (kotlin-reflect.jar) into the classpath.

-no-stdlib (JVM)

Don't automatically include the Kotlin/JVM stdlib (kotlin-stdlib.jar) and Kotlin reflection (kotlin-reflect.jar) into the classpath.

-script-templates classnames[,]

Script definition template classes. Use fully qualified class names and separate them with commas (,).

Kotlin/JS compiler options

The Kotlin compiler for JS compiles Kotlin source files into JavaScript code. The command-line tool for Kotlin to JS compilation is kotlinc-js.

In addition to the common options, Kotlin/JS compiler has the options listed below.

-libraries path

Paths to Kotlin libraries with .meta.js and .kjsm files, separated by the system path separator.

-main {call|noCall}

Define whether the main function should be called upon execution.

-meta-info

Generate .meta.js and .kjsm files with metadata. Use this option when creating a JS library.

-module-kind {umd|commonjs|amd|plain}

The kind of JS module generated by the compiler:

- umd - a Universal Module Definition module
- commonjs - a CommonJS module
- amd - an Asynchronous Module Definition module
- plain - a plain JS module

To learn more about the different kinds of JS module and the distinctions between them, see this article.

-no-stdlib (JS)

Don't automatically include the default Kotlin/JS stdlib into the compilation dependencies.

-output filepath

Set the destination file for the compilation result. The value must be a path to a .js file including its name.

-output-postfix filepath

Add the content of the specified file to the end of the output file.

-output-prefix filepath

Add the content of the specified file to the beginning of the output file.

-source-map

Generate the source map.

-source-map-base-dirs path

Use the specified paths as base directories. Base directories are used for calculating relative paths in the source map.

-source-map-embed-sources {always|never|inlining}

Embed source files into the source map.

-source-map-names-policy {simple-names|fully-qualified-names|no}

Add variable and function names that you declared in Kotlin code into the source map.

Setting	Description	Example output
simple-names	Variable names and simple function names are added. (Default)	main
fully-qualified-names	Variable names and fully qualified function names are added.	com.example.kjs.playground.main
no	No variable or function names are added.	N/A

-source-map-prefix

Add the specified prefix to paths in the source map.

Kotlin/Native compiler options

Kotlin/Native compiler compiles Kotlin source files into native binaries for the supported platforms. The command-line tool for Kotlin/Native compilation is kotlinc-native.

In addition to the common options, Kotlin/Native compiler has the options listed below.

-enable-assertions (-ea)

Enable runtime assertions in the generated code.

-g
Enable emitting debug information.

-generate-test-runner (-tr)
Produce an application for running unit tests from the project.

-generate-worker-test-runner (-trw)
Produce an application for running unit tests in a worker thread.

-generate-no-exit-test-runner (-trn)
Produce an application for running unit tests without an explicit process exit.

-include-binary path (-ib path)
Pack external binary within the generated klib file.

-library path (-l path)
Link with the library. To learn about using libraries in Kotlin/native projects, see Kotlin/Native libraries.

-library-version version (-lv version)
Set the library version.

-list-targets
List the available hardware targets.

-manifest path
Provide a manifest addend file.

-module-name name (Native)
Specify a name for the compilation module. This option can also be used to specify a name prefix for the declarations exported to Objective-C: How do I specify a custom Objective-C prefix/name for my Kotlin framework?

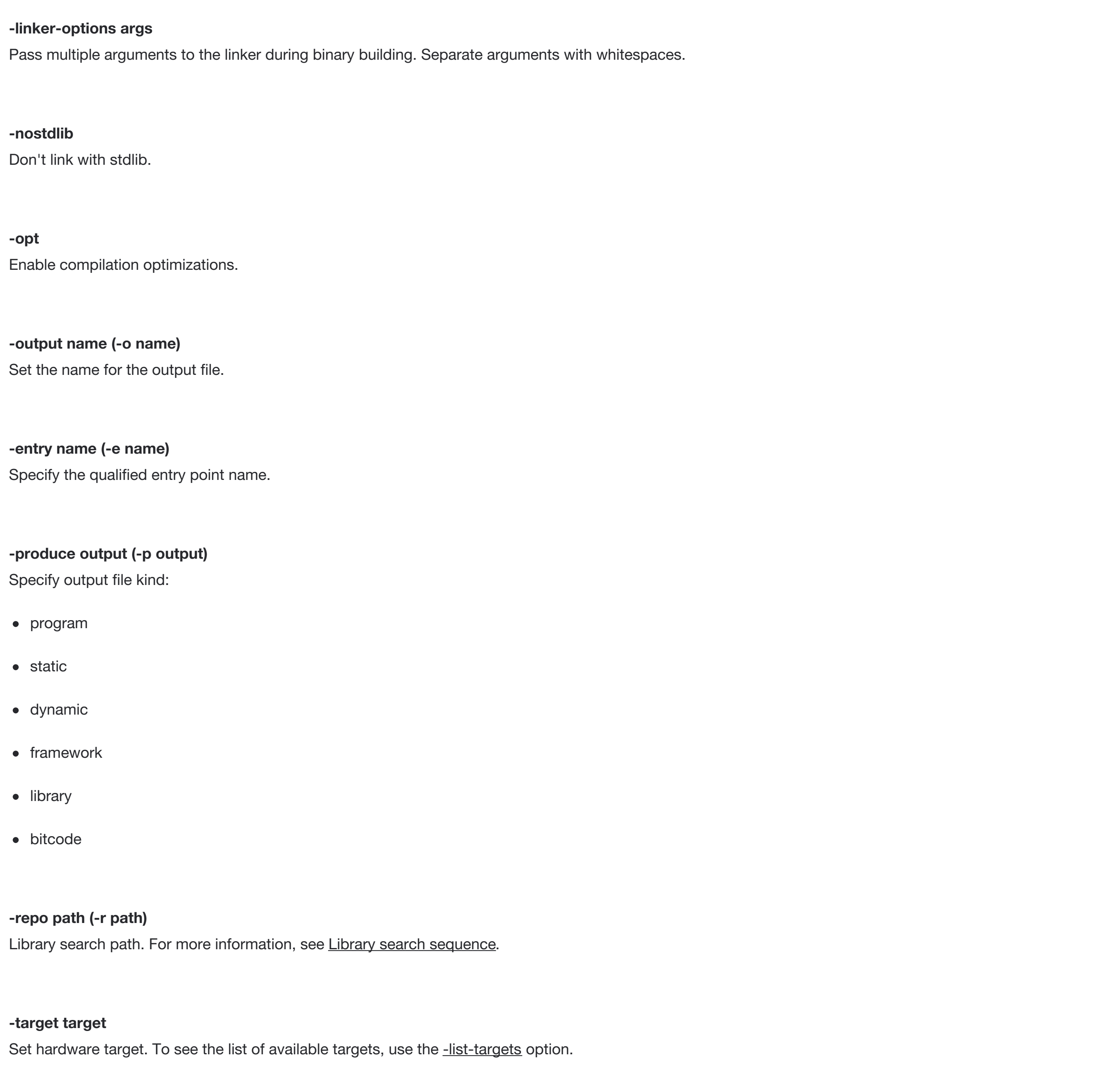
-native-library path (-nl path)
Include the native bitcode library.

-no-default-libs
Disable linking user code with the default platform libraries distributed with the compiler.

-nomain
Assume the main entry point to be provided by external libraries.

-nopack
Don't pack the library into a klib file.

-linker-option



Pass an argument to the linker during binary building. This can be used for linking against some native library.

-linker-options args
Pass multiple arguments to the linker during binary building. Separate arguments with whitespaces.

-nostdlib
Don't link with stdlib.

-opt
Enable compilation optimizations.

-output name (-o name)
Set the name for the output file.

-entry name (-e name)
Specify the qualified entry point name.

-produce output (-p output)
Specify output file kind:

- program
- static
- dynamic
- framework
- library
- bitcode

-repo path (-r path)
Library search path. For more information, see Library search sequence.

-target target
Set hardware target. To see the list of available targets, use the -list-targets option.

All-open compiler plugin

Kotlin has classes and their members final by default, which makes it inconvenient to use frameworks and libraries such as Spring AOP that require classes to be open. The all-open compiler plugin adapts Kotlin to the requirements of those frameworks and makes classes annotated with a specific annotation and their members open without the explicit open keyword.

For instance, when you use Spring, you don't need all the classes to be open, but only classes annotated with specific annotations like @Configuration or @Service. All-open allows to specify such annotations.

We provide all-open plugin support both for Gradle and Maven with the complete IDE integration.

> For Spring, you can use the kotlin-spring compiler plugin (see below).

Gradle

Add the plugin using Gradle's plugins DSL:

```
plugins {
    id "org.jetbrains.kotlin.plugin.allopen" version "1.9.0"
}
```

Then specify the list of annotations that will make classes open:

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation", "com.third.Annotation")
}
```

If the class (or any of its superclasses) is annotated with com.my.Annotation, the class itself and all its members will become open.

It also works with meta-annotations:

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // will be all-open
```

MyFrameworkAnnotation is annotated with the all-open meta-annotation com.my.Annotation, so it becomes an all-open annotation as well.

Maven

Here's how to use all-open with Maven:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- Or "spring" for the Spring support -->
            <plugin>all-open</plugin>
        </compilerPlugins>

        <pluginOptions>
            <!-- Each annotation is placed on its own line -->
            <option>all-open:annotation=com.my.Annotation</option>
            <option>all-open:annotation=com.their.AnotherAnnotation</option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-allopen</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Please refer to the Gradle section for the detailed information about how all-open annotations work.

Spring support

If you use Spring, you can enable the kotlin-spring compiler plugin instead of specifying Spring annotations manually. The kotlin-spring is a wrapper on top of all-open, and it behaves exactly the same way.

Add the plugin using Gradle's plugins DSL:

```
plugins {
    id "org.jetbrains.kotlin.plugin.spring" version "1.9.0"
}
```

In Maven, the spring plugin is provided by the kotlin-maven-allopen plugin dependency, so to enable it:

```
<compilerPlugins>
    <plugin>spring</plugin>
</compilerPlugins>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-allopen</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

The plugin specifies the following annotations:

- @Component
- @Async
- @Transactional
- @Cacheable
- @SpringBootTest

Thanks to meta-annotations support, classes annotated with @Configuration, @Controller, @RestController, @Service or @Repository are automatically opened since these annotations are meta-annotated with @Component.

Of course, you can use both kotlin-allopen and kotlin-spring in the same project.

Note that if you use the project template generated by the start.spring.io service, the kotlin-spring plugin will be enabled by default.

Command-line compiler

All-open compiler plugin JAR is available in the binary distribution of the Kotlin compiler. You can attach the plugin by providing the path to its JAR file using the Xplugin kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/allopen-compiler-plugin.jar
```

You can specify all-open annotations directly, using the annotation plugin option, or enable the "preset". The presets available now for all-open are spring, micronaut, and quarkus.

```
# The plugin option format is: "-P plugin:<plugin id>:<key>=<value>".
# Options can be repeated.

-P plugin:org.jetbrains.kotlin.allopen:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.allopen:preset=spring
```

No-arg compiler plugin

The no-arg compiler plugin generates an additional zero-argument constructor for classes with a specific annotation.

The generated constructor is synthetic so it can't be directly called from Java or Kotlin, but it can be called using reflection.

This allows the Java Persistence API (JPA) to instantiate a class although it doesn't have the zero-parameter constructor from Kotlin or Java point of view (see the description of kotlin-jpa plugin below).

Gradle

Add the plugin using Gradle's plugins DSL:

```
plugins {
  id "org.jetbrains.kotlin.plugin.noarg" version "1.9.0"
}
```

Then specify the list of no-arg annotations that must lead to generating a no-arg constructor for the annotated classes:

```
noArg {
    annotation("com.my.Annotation")
}
```

Enable invokeInitializers option if you want the plugin to run the initialization logic from the synthetic constructor. By default, it is disabled.

```
noArg {
    invokeInitializers = true
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- Or "jpa" for JPA support -->
            <plugin>no-arg</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>no-arg:annotation=com.my.Annotation</option>
            <!-- Call instance initializers in the synthetic constructor -->
            <!-- <option>no-arg:invokeInitializers=true</option> -->
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-noarg</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

JPA support

As with the kotlin-spring plugin wrapped on top of all-open, kotlin-jpa is wrapped on top of no-arg. The plugin specifies @Entity, @Embeddable, and @MappedSuperclass no-arg annotations automatically.

Add the plugin using the Gradle plugins DSL:

```
plugins {
  id "org.jetbrains.kotlin.plugin.jpa" version "1.9.0"
}
```

In Maven, enable the jpa plugin:

```
<compilerPlugins>
    <plugin>jpa</plugin>
</compilerPlugins>
```

Command-line compiler

Add the plugin JAR file to the compiler plugin classpath and specify annotations or presets:

```
-Xplugin=$KOTLIN_HOME/lib/noarg-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.noarg:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.noarg:preset=jpa
```

SAM-with-receiver compiler plugin

The sam-with-receiver compiler plugin makes the first parameter of the annotated Java "single abstract method" (SAM) interface method a receiver in Kotlin. This conversion only works when the SAM interface is passed as a Kotlin lambda, both for SAM adapters and SAM constructors (see the SAM conversions documentation for more details).

Here is an example:

```
public @interface SamWithReceiver {}

@SamWithReceiver
public interface TaskRunner {
    void run(Task task);
}
```

```
fun test(context: TaskContext) {
    val runner = TaskRunner {
        // Here 'this' is an instance of 'Task'

        println("$name is started")
        context.executeTask(this)
        println("$name is finished")
    }
}
```

Gradle

The usage is the same to all-open and no-arg, except the fact that sam-with-receiver does not have any built-in presets, and you need to specify your own list of special-treated annotations.

```
plugins {
    id("org.jetbrains.kotlin.plugin.sam.with.receiver") version "$kotlin_version"
}
```

Then specify the list of SAM-with-receiver annotations:

```
samWithReceiver {
    annotation("com.my.SamWithReceiver")
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <plugin>sam-with-receiver</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>
```

```
                sam-with-receiver:annotation=com.my.SamWithReceiver
            </option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-sam-with-receiver</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Command-line compiler

Add the plugin JAR file to the compiler plugin classpath and specify the list of sam-with-receiver annotations:

```
-Xplugin=$KOTLIN_HOME/lib/sam-with-receiver-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.samWithReceiver:annotation=com.my.SamWithReceiver
```

kapt compiler plugin

> kapt is in maintenance mode. We are keeping it up-to-date with recent Kotlin and Java releases but have no plans to implement new features. Please use the Kotlin Symbol Processing API (KSP) for annotation processing. See the list of libraries supported by KSP.

Annotation processors (see JSR 269) are supported in Kotlin with the kapt compiler plugin.

In a nutshell, you can use libraries such as Dagger or Data Binding in your Kotlin projects.

Please read below about how to apply the kapt plugin to your Gradle/Maven build.

Using in Gradle

Follow these steps:

1. Apply the kotlin-kapt Gradle plugin:

   Kotlin

   ```
   plugins {
       kotlin("kapt") version "1.9.0"
   }
   ```

   Groovy

   ```
   plugins {
       id "org.jetbrains.kotlin.kapt" version "1.9.0"
   }
   ```

2. Add the respective dependencies using the kapt configuration in your dependencies block:

   Kotlin

   ```
   dependencies {
       kapt("groupId:artifactId:version")
   }
   ```

Groovy

```
dependencies {
    kapt 'groupId:artifactId:version'
}
```

3. If you previously used the Android support for annotation processors, replace usages of the annotationProcessor configuration with kapt. If your project contains Java classes, kapt will also take care of them.

   If you use annotation processors for your androidTest or test sources, the respective kapt configurations are named kaptAndroidTest and kaptTest. Note that kaptAndroidTest and kaptTest extends kapt, so you can just provide the kapt dependency and it will be available both for production sources and tests.

Annotation processor arguments

Use arguments {} block to pass arguments to annotation processors:

```
kapt {
    arguments {
        arg("key", "value")
    }
}
```

Gradle build cache support

The kapt annotation processing tasks are cached in Gradle by default. However, annotation processors run arbitrary code that may not necessarily transform the task inputs into the outputs, might access and modify the files that are not tracked by Gradle etc. If the annotation processors used in the build cannot be properly cached, it is possible to disable caching for kapt entirely by adding the following lines to the build script, in order to avoid false-positive cache hits for the kapt tasks:

```
kapt {
    useBuildCache = false
}
```

Improving the speed of builds that use kapt

Running kapt tasks in parallel

To improve the speed of builds that use kapt, you can enable the Gradle Worker API for kapt tasks. Using the Worker API lets Gradle run independent annotation processing tasks from a single project in parallel, which in some cases significantly decreases the execution time.

When you use the custom JDK home feature in the Kotlin Gradle plugin, kapt task workers use only process isolation mode. Note that the kapt.workers.isolation property is ignored.

If you want to provide additional JVM arguments for a kapt worker process, use the input kaptProcessJvmArgs of the KaptWithoutKotlincTask:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.internal.KaptWithoutKotlincTask>()
    .configureEach {
        kaptProcessJvmArgs.add("-Xmx512m")
    }
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.internal.KaptWithoutKotlincTask.class)
    .configureEach {
        kaptProcessJvmArgs.add('-Xmx512m')
    }
```

Caching for annotation processors' classloaders

> Caching for annotation processors' classloaders in kapt is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Caching for annotation processors' classloaders helps kapt perform faster if you run many Gradle tasks consecutively.

To enable this feature, use the following properties in your gradle.properties file:

```
# positive value will enable caching
# use the same value as the number of modules that use kapt
kapt.classloaders.cache.size=5

# disable for caching to work
kapt.include.compile.classpath=false
```

If you run into any problems with caching for annotation processors, disable caching for them:

```
# specify annotation processors' full names to disable caching for them
kapt.classloaders.cache.disableForProcessors=[annotation processors full names]
```

Measuring performance of annotation processors

Get a performance statistics on the annotation processors execution using the -Kapt-show-processor-timings plugin option. An example output:

```
Kapt Annotation Processing performance report:
com.example.processor.TestingProcessor: total: 133 ms, init: 36 ms, 2 round(s): 97 ms, 0 ms
com.example.processor.AnotherProcessor: total: 100 ms, init: 6 ms, 1 round(s): 93 ms
```

You can dump this report into a file with the plugin option -Kapt-dump-processor-timings (org.jetbrains.kotlin.kapt3:dumpProcessorTimings). The following command will run kapt and dump the statistics to the ap-perf-report.file file:

```
kotlinc -cp $MY_CLASSPATH \
-Xplugin=kotlin-annotation-processing-SNAPSHOT.jar -P \
plugin:org.jetbrains.kotlin.kapt3:aptMode=stubsAndApt,\
plugin:org.jetbrains.kotlin.kapt3:apclasspath=processor/build/libs/processor.jar,\
plugin:org.jetbrains.kotlin.kapt3:dumpProcessorTimings=ap-perf-report.file \
-Xplugin=$JAVA_HOME/lib/tools.jar \
-d cli-tests/out \
-no-jdk -no-reflect -no-stdlib -verbose \
sample/src/main/
```

Measuring the number of files generated with annotation processors

The kotlin-kapt Gradle plugin can report statistics on the number of generated files for each annotation processor.

This is useful to track if there are unused annotation processors as a part of the build. You can use the generated report to find modules that trigger unnecessary annotation processors and update the modules to prevent that.

Enable the statistics in two steps:

- Set the showProcessorStats flag to true in your build.gradle(.kts):

```
kapt {
    showProcessorStats = true
}
```

- Set the kapt.verbose Gradle property to true in your gradle.properties:

```
kapt.verbose=true
```

> You can also enable verbose output via the command line option verbose.

The statistics will appear in the logs with the info level. You'll see the Annotation processor stats: line followed by statistics on the execution time of each annotation processor. After these lines, there will be the Generated files report: line followed by statistics on the number of generated files for each annotation processor. For example:

```
[INFO] Annotation processor stats:
[INFO] org.mapstruct.ap.MappingProcessor: total: 290 ms, init: 1 ms, 3 round(s): 289 ms, 0 ms, 0 ms
[INFO] Generated files report:
[INFO] org.mapstruct.ap.MappingProcessor: total sources: 2, sources per round: 2, 0, 0
```

Compile avoidance for kapt

To improve the times of incremental builds with kapt, it can use the Gradle compile avoidance. With compile avoidance enabled, Gradle can skip annotation processing when rebuilding a project. Particularly, annotation processing is skipped when:

- The project's source files are unchanged.
- The changes in dependencies are ABI compatible. For example, the only changes are in method bodies.

However, compile avoidance can't be used for annotation processors discovered in the compile classpath since any changes in them require running the annotation processing tasks.

To run kapt with compile avoidance:

- Add the annotation processor dependencies to the kapt* configurations manually as described above.
- Turn off the discovery of annotation processors in the compile classpath by adding this line to your gradle.properties file:

```
kapt.include.compile.classpath=false
```

Incremental annotation processing

kapt supports incremental annotation processing that is enabled by default. Currently, annotation processing can be incremental only if all annotation processors being used are incremental.

To disable incremental annotation processing, add this line to your gradle.properties file:

```
kapt.incremental.apt=false
```

Note that incremental annotation processing requires incremental compilation to be enabled as well.

Java compiler options

kapt uses Java compiler to run annotation processors.
Here is how you can pass arbitrary options to javac:

```
kapt {
    javacOptions {
        // Increase the max count of errors from annotation processors.
        // Default is 100.
        option("-Xmaxerrs", 500)
    }
}
```

Non-existent type correction

Some annotation processors (such as AutoFactory) rely on precise types in declaration signatures. By default, kapt replaces every unknown type (including types for

the generated classes) to NonExistentClass, but you can change this behavior. Add the option to the build.gradle(.kts) file to enable error type inferring in stubs:

```
kapt {
    correctErrorTypes = true
}
```

Using in Maven

Add an execution of the kapt goal from kotlin-maven-plugin before compile:

```
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal> <!-- You can skip the <goals> element
        if you enable extensions for the plugin -->
    </goals>
    <configuration>
        <sourceDirs>
            <sourceDir>src/main/kotlin</sourceDir>
            <sourceDir>src/main/java</sourceDir>
        </sourceDirs>
        <annotationProcessorPaths>
            <!-- Specify your annotation processors here -->
            <annotationProcessorPath>
                <groupId>com.google.dagger</groupId>
                <artifactId>dagger-compiler</artifactId>
                <version>2.9</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</execution>
```

To configure the level of annotation processing, set one of the following as the aptMode in the <configuration> block:

- stubs – only generate stubs needed for annotation processing.
- apt – only run annotation processing.
- stubsAndApt – (default) generate stubs and run annotation processing.

For example:

```
<configuration>
   ...
   <aptMode>stubs</aptMode>
</configuration>
```

Using in IntelliJ build system

kapt is not supported for IntelliJ IDEA's own build system. Launch the build from the "Maven Projects" toolbar whenever you want to re-run the annotation processing.

Using in CLI

kapt compiler plugin is available in the binary distribution of the Kotlin compiler.

You can attach the plugin by providing the path to its JAR file using the Xplugin kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/kotlin-annotation-processing.jar
```

Here is a list of the available options:

- sources (required): An output path for the generated files.
- classes (required): An output path for the generated class files and resources.

- stubs (required): An output path for the stub files. In other words, some temporary directory.
- incrementalData: An output path for the binary stubs.
- apclasspath (repeatable): A path to the annotation processor JAR. Pass as many apclasspath options as many JARs you have.
- apoptions: A base64-encoded list of the annotation processor options. See AP/javac options encoding for more information.
- javacArguments: A base64-encoded list of the options passed to javac. See AP/javac options encoding for more information.
- processors: A comma-specified list of annotation processor qualified class names. If specified, kapt does not try to find annotation processors in apclasspath.
- verbose: Enable verbose output.
- aptMode (required)
  - stubs – only generate stubs needed for annotation processing.
  - apt – only run annotation processing.
  - stubsAndApt – generate stubs and run annotation processing.
- correctErrorTypes: See below. Disabled by default.

The plugin option format is: -P plugin:<plugin id>:<key>=<value>. Options can be repeated.

An example:

```
-P plugin:org.jetbrains.kotlin.kapt3:sources=build/kapt/sources
-P plugin:org.jetbrains.kotlin.kapt3:classes=build/kapt/classes
-P plugin:org.jetbrains.kotlin.kapt3:stubs=build/kapt/stubs

-P plugin:org.jetbrains.kotlin.kapt3:apclasspath=lib/ap.jar
-P plugin:org.jetbrains.kotlin.kapt3:apclasspath=lib/anotherAp.jar

-P plugin:org.jetbrains.kotlin.kapt3:correctErrorTypes=true
```

Generating Kotlin sources

kapt can generate Kotlin sources. Just write the generated Kotlin source files to the directory specified by processingEnv.options["kapt.kotlin.generated"], and these files will be compiled together with the main sources.

Note that kapt does not support multiple rounds for the generated Kotlin files.

AP/Javac options encoding

apoptions and javacArguments CLI options accept an encoded map of options.
Here is how you can encode options by yourself:

```
fun encodeList(options: Map<String, String>): String {
    val os = ByteArrayOutputStream()
    val oos = ObjectOutputStream(os)

    oos.writeInt(options.size)
    for ((key, value) in options.entries) {
        oos.writeUTF(key)
        oos.writeUTF(value)
    }

    oos.flush()
    return Base64.getEncoder().encodeToString(os.toByteArray())
}
```

Keeping Java compiler's annotation processors

By default, kapt runs all annotation processors and disables annotation processing by javac. However, you may need some of javac's annotation processors working (for example, Lombok).

In the Gradle build file, use the option keepJavacAnnotationProcessors:

```
kapt {
    keepJavacAnnotationProcessors = true
}
```

If you use Maven, you need to specify concrete plugin settings. See this example of settings for the Lombok compiler plugin.

Lombok compiler plugin

> The Lombok compiler plugin is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The Kotlin Lombok compiler plugin allows the generation and use of Java's Lombok declarations by Kotlin code in the same mixed Java/Kotlin module. If you call such declarations from another module, then you don't need to use this plugin for the compilation of that module.

The Lombok compiler plugin cannot replace Lombok, but it helps Lombok work in mixed Java/Kotlin modules. Thus, you still need to configure Lombok as usual when using this plugin. Learn more about how to configure the Lombok compiler plugin.

Supported annotations

The plugin supports the following annotations:

- @Getter, @Setter
- @Builder
- @NoArgsConstructor, @RequiredArgsConstructor, and @AllArgsConstructor
- @Data
- @With
- @Value

We're continuing to work on this plugin. To find out the detailed current state, visit the Lombok compiler plugin's README.

Currently, we don't have plans to support the @SuperBuilder and @Tolerate annotations. However, we can consider this if you vote for @SuperBuilder and @Tolerate in YouTrack.

> Kotlin compiler ignores Lombok annotations if you use them in Kotlin code.

Gradle

Apply the kotlin-plugin-lombok Gradle plugin in the build.gradle(.kts) file:

Kotlin

```
plugins {
    kotlin("plugin.lombok") version "1.9.0"
    id("io.freefair.lombok") version "5.3.0"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.plugin.lombok' version '1.9.0'
```

```
    id 'io.freefair.lombok' version '5.3.0'
}
```

See this test project with examples of the Lombok compiler plugin in use.

Using the Lombok configuration file

If you use a Lombok configuration file lombok.config, you need to set the file's path so that the plugin can find it. The path must be relative to the module's directory. For example, add the following code to your build.gradle(.kts) file:

Kotlin

```kotlin
kotlinLombok {
    lombokConfigurationFile(file("lombok.config"))
}
```

Groovy

```groovy
kotlinLombok {
    lombokConfigurationFile file("lombok.config")
}
```

See this test project with examples of the Lombok compiler plugin and lombok.config in use.

Maven

To use the Lombok compiler plugin, add the plugin lombok to the compilerPlugins section and the dependency kotlin-maven-lombok to the dependencies section. If you use a Lombok configuration file lombok.config, provide a path to it to the plugin in the pluginOptions. Add the following lines to the pom.xml file:

```xml
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <configuration>
        <compilerPlugins>
            <plugin>lombok</plugin>
        </compilerPlugins>
        <pluginOptions>
            <option>lombok:config=${project.basedir}/lombok.config</option>
        </pluginOptions>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-lombok</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.20</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</plugin>
```

See this test project example of the Lombok compiler plugin and lombok.config in use.

Using with kapt

By default, the kapt compiler plugin runs all annotation processors and disables annotation processing by javac. To run Lombok along with kapt, set up kapt to keep javac's annotation processors working.

If you use Gradle, add the option to the build.gradle(.kts) file:

```
kapt {
    keepJavacAnnotationProcessors = true
}
```

In Maven, use the following settings to launch Lombok with Java's compiler:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>${lombok.version}</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</plugin>
```

The Lombok compiler plugin works correctly with kapt if annotation processors don't depend on the code generated by Lombok.

Look through the test project examples of kapt and the Lombok compiler plugin in use:

- Using Gradle.
- Using Maven

Command-line compiler

Lombok compiler plugin JAR is available in the binary distribution of the Kotlin compiler. You can attach the plugin by providing the path to its JAR file using the Xplugin kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/lombok-compiler-plugin.jar
```

If you want to use the lombok.config file, replace <PATH_TO_CONFIG_FILE> with a path to your lombok.config:

```
# The plugin option format is: "-P plugin:<plugin id>:<key>=<value>".
# Options can be repeated.

-P plugin:org.jetbrains.kotlin.lombok:config=<PATH_TO_CONFIG_FILE>
```

Kotlin Symbol Processing API

Kotlin Symbol Processing (KSP) is an API that you can use to develop lightweight compiler plugins. KSP provides a simplified compiler plugin API that leverages the power of Kotlin while keeping the learning curve at a minimum. Compared to kapt, annotation processors that use KSP can run up to 2 times faster.

To learn more about how KSP compares to kapt, check out why KSP. To get started writing a KSP processor, take a look at the KSP quickstart.

Overview

The KSP API processes Kotlin programs idiomatically. KSP understands Kotlin-specific features, such as extension functions, declaration-site variance, and local functions. It also models types explicitly and provides basic type checking, such as equivalence and assign-compatibility.

The API models Kotlin program structures at the symbol level according to Kotlin grammar. When KSP-based plugins process source programs, constructs like classes, class members, functions, and associated parameters are accessible for the processors, while things like if blocks and for loops are not.

Conceptually, KSP is similar to KType in Kotlin reflection. The API allows processors to navigate from class declarations to corresponding types with specific type arguments and vice-versa. You can also substitute type arguments, specify variances, apply star projections, and mark nullabilities of types.

Another way to think of KSP is as a preprocessor framework of Kotlin programs. By considering KSP-based plugins as symbol processors, or simply processors, the data flow in a compilation can be described in the following steps:

1. Processors read and analyze source programs and resources.
2. Processors generate code or other forms of output.
3. The Kotlin compiler compiles the source programs together with the generated code.

Unlike a full-fledged compiler plugin, processors cannot modify the code. A compiler plugin that changes language semantics can sometimes be very confusing. KSP avoids that by treating the source programs as read-only.

You can also get an overview of KSP in this video:

Watch video online.

How KSP looks at source files

Most processors navigate through the various program structures of the input source code. Before diving into usage of the API, let's see at how a file might look from KSP's point of view:

```
KSFile
  packageName: KSName
  fileName: String
  annotations: List<KSAnnotation>  (File annotations)
  declarations: List<KSDeclaration>
    KSClassDeclaration // class, interface, object
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      classKind: ClassKind
      primaryConstructor: KSFunctionDeclaration
      superTypes: List<KSTypeReference>
      // contains inner classes, member functions, properties, etc.
      declarations: List<KSDeclaration>
    KSFunctionDeclaration // top level function
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      functionKind: FunctionKind
      extensionReceiver: KSTypeReference?
      returnType: KSTypeReference
      parameters: List<KSValueParameter>
      // contains local classes, local functions, local variables, etc.
      declarations: List<KSDeclaration>
    KSPropertyDeclaration // global variable
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
```

```
        typeParameters: KSTypeParameter
        parentDeclaration: KSDeclaration
        extensionReceiver: KSTypeReference?
        type: KSTypeReference
        getter: KSPropertyGetter
          returnType: KSTypeReference
        setter: KSPropertySetter
          parameter: KSValueParameter
```

This view lists common things that are declared in the file: classes, functions, properties, and so on.

SymbolProcessorProvider: the entry point

KSP expects an implementation of the SymbolProcessorProvider interface to instantiate SymbolProcessor:

```
interface SymbolProcessorProvider {
    fun create(environment: SymbolProcessorEnvironment): SymbolProcessor
}
```

While SymbolProcessor is defined as:

```
interface SymbolProcessor {
    fun process(resolver: Resolver): List<KSAnnotated> // Let's focus on this
    fun finish() {}
    fun onError() {}
}
```

A Resolver provides SymbolProcessor with access to compiler details such as symbols. A processor that finds all top-level functions and non-local functions in top-level classes might look something like the following:

```
class HelloFunctionFinderProcessor : SymbolProcessor() {
    // ...
    val functions = mutableListOf<KSClassDeclaration>()
    val visitor = FindFunctionsVisitor()

    override fun process(resolver: Resolver) {
        resolver.getAllFiles().forEach { it.accept(visitor, Unit) }
    }

    inner class FindFunctionsVisitor : KSVisitorVoid() {
        override fun visitClassDeclaration(classDeclaration: KSClassDeclaration, data: Unit) {
            classDeclaration.getDeclaredFunctions().forEach { it.accept(this, Unit) }
        }

        override fun visitFunctionDeclaration(function: KSFunctionDeclaration, data: Unit) {
            functions.add(function)
        }

        override fun visitFile(file: KSFile, data: Unit) {
            file.declarations.forEach { it.accept(this, Unit) }
        }
    }
    // ...

    class Provider : SymbolProcessorProvider {
        override fun create(environment: SymbolProcessorEnvironment): SymbolProcessor = TODO()
    }
}
```

Resources

- Quickstart
- Why use KSP?
- Examples
- How KSP models Kotlin code

- Reference for Java annotation processor authors
- Incremental processing notes
- Multiple round processing notes
- KSP on multiplatform projects
- Running KSP from command line
- FAQ

Supported libraries

The table below includes a list of popular libraries on Android and their various stages of support for KSP.

Library	Status	Tracking issue for KSP
Room	Officially supported	
Moshi	Officially supported	
RxHttp	Officially supported	
Kotshi	Officially supported	
Lyricist	Officially supported	
Lich SavedState	Officially supported	
gRPC Dekorator	Officially supported	
EasyAdapter	Officially supported	
Koin Annotations	Officially supported	
Auto Factory	Not yet supported	Link
Dagger	Not yet supported	Link
Hilt	Not yet supported	Link
Glide	Officially supported	
DeeplinkDispatch	Supported via airbnb/DeepLinkDispatch#323	

Library	Status	Tracking issue for KSP
Micronaut	In Progress	Link
Epoxy	Officially supported	
Paris	Officially supported	
Auto Dagger	Officially supported	
SealedX	Officially supported	

KSP quickstart

For a quick start, you can create your own processor or get a sample one.

Create a processor of your own

1. Create an empty gradle project.
2. Specify version 1.9.0 of the Kotlin plugin in the root project for use in other project modules:

Kotlin

```
plugins {
    kotlin("jvm") version "1.9.0" apply false
}

buildscript {
    dependencies {
        classpath(kotlin("gradle-plugin", version = "1.9.0"))
    }
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.9.0' apply false
}

buildscript {
    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:1.9.0'
    }
}
```

3. Add a module for hosting the processor.
4. In the module's build script, apply Kotlin plugin and add the KSP API to the dependencies block.

Kotlin

```
plugins {
    kotlin("jvm")
}
```

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("com.google.devtools.ksp:symbol-processing-api:1.9.0-1.0.11")
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.9.0'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.google.devtools.ksp:symbol-processing-api:1.9.0-1.0.11'
}
```

5. You'll need to implement com.google.devtools.ksp.processing.SymbolProcessor and com.google.devtools.ksp.processing.SymbolProcessorProvider. Your implementation of SymbolProcessorProvider will be loaded as a service to instantiate the SymbolProcessor you implement. Note the following:

   - Implement SymbolProcessorProvider.create() to create a SymbolProcessor. Pass dependencies that your processor needs (such as CodeGenerator, processor options) through the parameters of SymbolProcessorProvider.create().
   - Your main logic should be in the SymbolProcessor.process() method.
   - Use resolver.getSymbolsWithAnnotation() to get the symbols you want to process, given the fully-qualified name of an annotation.
   - A common use case for KSP is to implement a customized visitor (interface com.google.devtools.ksp.symbol.KSVisitor) for operating on symbols. A simple template visitor is com.google.devtools.ksp.symbol.KSDefaultVisitor.
   - For sample implementations of the SymbolProcessorProvider and SymbolProcessor interfaces, see the following files in the sample project.
     - src/main/kotlin/BuilderProcessor.kt
     - src/main/kotlin/TestProcessor.kt
   - After writing your own processor, register your processor provider to the package by including its fully-qualified name in resources/META-INF/services/com.google.devtools.ksp.processing.SymbolProcessorProvider.

Use your own processor in a project

1. Create another module that contains a workload where you want to try out your processor.

Kotlin

```
pluginManagement {
    repositories {
        gradlePluginPortal()
    }
}
```

Groovy

```
pluginManagement {
    repositories {
        gradlePluginPortal()
    }
}
```

2. In the module's build script, apply the com.google.devtools.ksp plugin with the specified version and add your processor to the list of dependencies.

Kotlin

```kotlin
plugins {
    id("com.google.devtools.ksp") version "1.9.0-1.0.11"
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))
    implementation(project(":test-processor"))
    ksp(project(":test-processor"))
}
```

Groovy

```groovy
plugins {
    id 'com.google.devtools.ksp' version '1.9.0-1.0.11'
}

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version'
    implementation project(':test-processor')
    ksp project(':test-processor')
}
```

3. Run ./gradlew build. You can find the generated code under build/generated/source/ksp.

Here's a sample build script to apply the KSP plugin to a workload:

Kotlin

```kotlin
plugins {
    id("com.google.devtools.ksp") version "1.9.0-1.0.11"
    kotlin("jvm")
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))
    implementation(project(":test-processor"))
    ksp(project(":test-processor"))
}
```

Groovy

```groovy
plugins {
    id 'com.google.devtools.ksp' version '1.9.0-1.0.11'
    id 'org.jetbrains.kotlin.jvm' version '1.9.0'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib:1.9.0'
    implementation project(':test-processor')
    ksp project(':test-processor')
}
```

Pass options to processors

Processor options in SymbolProcessorEnvironment.options are specified in gradle build scripts:

```kotlin
ksp {
    arg("option1", "value1")
    arg("option2", "value2")
```

```
    ...
}
```

Make IDE aware of generated code

> Generated source files are registered automatically since KSP 1.8.0-1.0.9. If you're using KSP 1.0.9 or newer and don't need to make the IDE aware of generated resources, feel free to skip this section.

By default, IntelliJ IDEA or other IDEs don't know about the generated code. So it will mark references to generated symbols unresolvable. To make an IDE be able to reason about the generated symbols, mark the following paths as generated source roots:

```
build/generated/ksp/main/kotlin/
build/generated/ksp/main/java/
```

If your IDE supports resource directories, also mark the following one:

```
build/generated/ksp/main/resources/
```

It may also be necessary to configure these directories in your KSP consumer module's build script:

Kotlin

```
kotlin {
    sourceSets.main {
        kotlin.srcDir("build/generated/ksp/main/kotlin")
    }
    sourceSets.test {
        kotlin.srcDir("build/generated/ksp/test/kotlin")
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        main.kotlin.srcDirs += 'build/generated/ksp/main/kotlin'
        test.kotlin.srcDirs += 'build/generated/ksp/test/kotlin'
    }
}
```

If you are using IntelliJ IDEA and KSP in a Gradle plugin then the above snippet will give the following warning:

```
Execution optimizations have been disabled for task ':publishPluginJar' to ensure correctness due to the following reasons:
Gradle detected a problem with the following location: '../build/generated/ksp/main/kotlin'.
Reason: Task ':publishPluginJar' uses this output of task ':kspKotlin' without declaring an explicit or implicit dependency.
```

In this case, use the following script instead:

Kotlin

```
plugins {
    // ...
    idea
}

idea {
    module {
        // Not using += due to https://github.com/gradle/gradle/issues/8749
        sourceDirs = sourceDirs + file("build/generated/ksp/main/kotlin") // or tasks["kspKotlin"].destination
        testSourceDirs = testSourceDirs + file("build/generated/ksp/test/kotlin")
        generatedSourceDirs = generatedSourceDirs + file("build/generated/ksp/main/kotlin") +
```

```
file("build/generated/ksp/test/kotlin")
    }
}
```

Groovy

```
plugins {
    // ...
    id 'idea'
}

idea {
    module {
        // Not using += due to https://github.com/gradle/gradle/issues/8749
        sourceDirs = sourceDirs + file('build/generated/ksp/main/kotlin') // or tasks["kspKotlin"].destination
        testSourceDirs = testSourceDirs + file('build/generated/ksp/test/kotlin')
        generatedSourceDirs = generatedSourceDirs + file('build/generated/ksp/main/kotlin') +
file('build/generated/ksp/test/kotlin')
    }
}
```

Why KSP

Compiler plugins are powerful metaprogramming tools that can greatly enhance how you write code. Compiler plugins call compilers directly as libraries to analyze and edit input programs. These plugins can also generate output for various uses. For example, they can generate boilerplate code, and they can even generate full implementations for specially-marked program elements, such as Parcelable. Plugins have a variety of other uses and can even be used to implement and fine-tune features that are not provided directly in a language.

While compiler plugins are powerful, this power comes at a price. To write even the simplest plugin, you need to have some compiler background knowledge, as well as a certain level of familiarity with the implementation details of your specific compiler. Another practical issue is that plugins are often closely tied to specific compiler versions, meaning you might need to update your plugin each time you want to support a newer version of the compiler.

KSP makes creating lightweight compiler plugins easier

KSP is designed to hide compiler changes, minimizing maintenance efforts for processors that use it. KSP is designed not to be tied to the JVM so that it can be adapted to other platforms more easily in the future. KSP is also designed to minimize build times. For some processors, such as Glide, KSP reduces full compilation times by up to 25% when compared to kapt.

KSP is itself implemented as a compiler plugin. There are prebuilt packages on Google's Maven repository that you can download and use without having to build the project yourself.

Comparison to kotlinc compiler plugins

kotlinc compiler plugins have access to almost everything from the compiler and therefore have maximum power and flexibility. On the other hand, because these plugins can potentially depend on anything in the compiler, they are sensitive to compiler changes and need to be maintained frequently. These plugins also require a deep understanding of kotlinc's implementation, so the learning curve can be steep.

KSP aims to hide most compiler changes through a well-defined API, though major changes in compiler or even the Kotlin language might still require to be exposed to API users.

KSP tries to fulfill common use cases by providing an API that trades power for simplicity. Its capability is a strict subset of a general kotlinc plugin. For example, while kotlinc can examine expressions and statements and can even modify code, KSP cannot.

While writing a kotlinc plugin can be a lot of fun, it can also take a lot of time. If you aren't in a position to learn kotlinc's implementation and do not need to modify source code or read expressions, KSP might be a good fit.

Comparison to reflection

KSP's API looks similar to kotlin.reflect. The major difference between them is that type references in KSP need to be resolved explicitly. This is one of the reasons why the interfaces are not shared.

Comparison to kapt

kapt is a remarkable solution which makes a large amount of Java annotation processors work for Kotlin programs out-of-box. The major advantages of KSP over kapt are improved build performance, not tied to JVM, a more idiomatic Kotlin API, and the ability to understand Kotlin-only symbols.

To run Java annotation processors unmodified, kapt compiles Kotlin code into Java stubs that retain information that Java annotation processors care about. To create these stubs, kapt needs to resolve all symbols in the Kotlin program. The stub generation costs roughly 1/3 of a full kotlinc analysis and the same order of kotlinc code-generation. For many annotation processors, this is much longer than the time spent in the processors themselves. For example, Glide looks at a very limited number of classes with a predefined annotation, and its code generation is fairly quick. Almost all of the build overhead resides in the stub generation phase. Switching to KSP would immediately reduce the time spent in the compiler by 25%.

For performance evaluation, we implemented a simplified version of Glide in KSP to make it generate code for the Tachiyomi project. While the total Kotlin compilation time of the project is 21.55 seconds on our test device, it took 8.67 seconds for kapt to generate the code, and it took 1.15 seconds for our KSP implementation to generate the code.

Unlike kapt, processors in KSP do not see input programs from Java's point of view. The API is more natural to Kotlin, especially for Kotlin-specific features such as top-level functions. Because KSP doesn't delegate to javac like kapt, it doesn't assume JVM-specific behaviors and can be used with other platforms potentially.

Limitations

While KSP tries to be a simple solution for most common use cases, it has made several trade-offs compared to other plugin solutions. The following are not goals of KSP:

- Examining expression-level information of source code.
- Modifying source code.
- 100% compatibility with the Java Annotation Processing API.

We are also exploring several additional features. These features are currently unavailable:

- IDE integration: Currently IDEs know nothing about the generated code.

KSP examples

Get all member functions

```kotlin
fun KSClassDeclaration.getDeclaredFunctions(): List<KSFunctionDeclaration> =
    declarations.filterIsInstance<KSFunctionDeclaration>()
```

Check whether a class or function is local

```kotlin
fun KSDeclaration.isLocal(): Boolean =
    parentDeclaration != null && parentDeclaration !is KSClassDeclaration
```

Find the actual class or interface declaration that the type alias points to

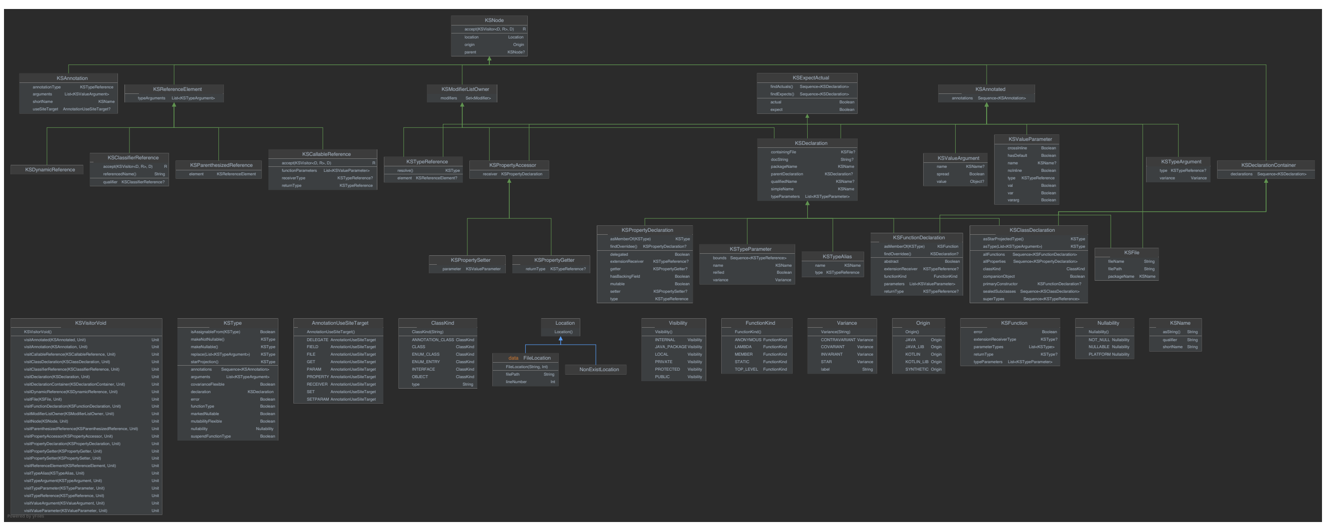
```kotlin
fun KSTypeAlias.findActualType(): KSClassDeclaration {
    val resolvedType = this.type.resolve().declaration
    return if (resolvedType is KSTypeAlias) {
        resolvedType.findActualType()
    } else {
        resolvedType as KSClassDeclaration
    }
}
```

Collect suppressed names in a file annotation

```
// @file:kotlin.Suppress("Example1", "Example2")
fun KSFile.suppressedNames(): List<String> {
    val ignoredNames = mutableListOf<String>()
    annotations.filter {
        it.shortName.asString() == "Suppress" && it.annotationType.resolve()?.declaration?.qualifiedName?.asString() ==
"kotlin.Suppress"
    }.forEach {
        val argValues: List<String> = it.arguments.flatMap { it.value }
        ignoredNames.addAll(argValues)
    }
    return ignoredNames
}
```

How KSP models Kotlin code

You can find the API definition in the KSP GitHub repository. The diagram shows an overview of how Kotlin is modeled in KSP:



class diagram

See the full-sized diagram.

Type and resolution

The resolution takes most of the cost of the underlying API implementation. So type references are designed to be resolved by processors explicitly (with a few exceptions). When a type (such as KSFunctionDeclaration.returnType or KSAnnotation.annotationType) is referenced, it is always a KSTypeReference, which is a KSReferenceElement with annotations and modifiers.

```
interface KSFunctionDeclaration : ... {
  val returnType: KSTypeReference?
  // ...
}

interface KSTypeReference : KSAnnotated, KSModifierListOwner {
  val type: KSReferenceElement
}
```

A KSTypeReference can be resolved to a KSType, which refers to a type in Kotlin's type system.

A KSTypeReference has a KSReferenceElement, which models Kotlin's program structure: namely, how the reference is written. It corresponds to the type element in Kotlin's grammar.

A KSReferenceElement can be a KSClassifierReference or KSCallableReference, which contains a lot of useful information without the need for resolution. For example, KSClassifierReference has referencedName, while KSCallableReference has receiverType, functionArguments, and returnType.

If the original declaration referenced by a KSTypeReference is needed, it can usually be found by resolving to KSType and accessing through KSType.declaration. Moving from where a type is mentioned to where its class is defined looks like this:

```
val ksType: KSType = ksTypeReference.resolve()
val ksDeclaration: KSDeclaration = ksType.declaration
```

Type resolution is costly and therefore has explicit form. Some of the information obtained from resolution is already available in KSReferenceElement. For example, KSClassifierReference.referencedName can filter out a lot of elements that are not interesting. You should resolve type only if you need specific information from KSDeclaration or KSType.

KSTypeReference pointing to a function type has most of its information in its element. Although it can be resolved to the family of Function0, Function1, and so on, these resolutions don't bring any more information than KSCallableReference. One use case for resolving function type references is dealing with the identity of the function's prototype.

Java annotation processing to KSP reference

Program elements

Java	Closest facility in KSP	Notes
AnnotationMirror	KSAnnotation	
AnnotationValue	KSValueArguments	
Element	KSDeclaration / KSDeclarationContainer	
ExecutableElement	KSFunctionDeclaration	
PackageElement	KSFile	KSP doesn't model packages as program elements
Parameterizable	KSDeclaration	
QualifiedNameable	KSDeclaration	
TypeElement	KSClassDeclaration	
TypeParameterElement	KSTypeParameter	
VariableElement	KSValueParameter / KSPropertyDeclaration	

Types

KSP requires explicit type resolution, so some functionalities in Java can only be carried out by KSType and the corresponding elements before resolution.

Java	Closest facility in KSP	Notes
ArrayType	KSBuiltIns.arrayType	
DeclaredType	KSType / KSClassifierReference	
ErrorType	KSType.isError	
ExecutableType	KSType / KSCallableReference	
IntersectionType	KSType / KSTypeParameter	
NoType	KSType.isError	N/A in KSP
NullType		N/A in KSP
PrimitiveType	KSBuiltIns	Not exactly same as primitive type in Java
ReferenceType	KSTypeReference	
TypeMirror	KSType	
TypeVariable	KSTypeParameter	
UnionType	N/A	Kotlin has only one type per catch block. UnionType is also not observable by even Java annotation processors
WildcardType	KSType / KSTypeArgument	

Misc

Java	Closest facility in KSP	Notes
Name	KSName	
ElementKind	ClassKind / FunctionKind	
Modifier	Modifier	

Java	Closest facility in KSP	Notes
NestingKind	ClassKind / FunctionKind	
AnnotationValueVisitor		
ElementVisitor	KSVisitor	
AnnotatedConstruct	KSAnnotated	
TypeVisitor		
TypeKind	KSBuiltIns	Some can be found in builtins, otherwise check KSClassDeclaration for DeclaredType
ElementFilter	Collection.filterIsInstance	
ElementKindVisitor	KSVisitor	
ElementScanner	KSTopDownVisitor	
SimpleAnnotationValueVisitor		Not needed in KSP
SimpleElementVisitor	KSVisitor	
SimpleTypeVisitor		
TypeKindVisitor		
Types	Resolver / utils	Some of the utils are also integrated into symbol interfaces
Elements	Resolver / utils	

Details

See how functionalities of Java annotation processing API can be carried out by KSP.

AnnotationMirror

Java	KSP equivalent

Java	KSP equivalent
getAnnotationType	ksAnnotation.annotationType
getElementValues	ksAnnotation.arguments

AnnotationValue

Java	KSP equivalent
getValue	ksValueArgument.value

Element

Java	KSP equivalent
asType	ksClassDeclaration.asType(...) is available for KSClassDeclaration only. Type arguments need to be supplied.
getAnnotation	To be implemented
getAnnotationMirrors	ksDeclaration.annotations
getEnclosedElements	ksDeclarationContainer.declarations
getEnclosingElements	ksDeclaration.parentDeclaration
getKind	Type check and cast following ClassKind or FunctionKind
getModifiers	ksDeclaration.modifiers
getSimpleName	ksDeclaration.simpleName

ExecutableElement

Java	KSP equivalent
getDefaultValue	To be implemented
getParameters	ksFunctionDeclaration.parameters

Java	KSP equivalent
getReceiverType	ksFunctionDeclaration.parentDeclaration
getReturnType	ksFunctionDeclaration.returnType
getSimpleName	ksFunctionDeclaration.simpleName
getThrownTypes	Not needed in Kotlin
getTypeParameters	ksFunctionDeclaration.typeParameters
isDefault	Check whether parent declaration is an interface or not
isVarArgs	ksFunctionDeclaration.parameters.any { it.isVarArg }

Parameterizable

Java	KSP equivalent
getTypeParameters	ksFunctionDeclaration.typeParameters

QualifiedNameable

Java	KSP equivalent
getQualifiedName	ksDeclaration.qualifiedName

TypeElement

Java	KSP equivalent
getEnclosedElements	ksClassDeclaration.declarations
getEnclosingElement	ksClassDeclaration.parentDeclaration
getInterfaces	(code below)

```
// Should be able to do without resolution
ksClassDeclaration.superTypes
    .map { it.resolve() }
    .filter { (it?.declaration as? KSClassDeclaration)?.classKind == ClassKind.INTERFACE
}
```

Java	KSP equivalent
getNestingKind	Check KSClassDeclaration.parentDeclaration and inner modifier
getQualifiedName	ksClassDeclaration.qualifiedName
getSimpleName	ksClassDeclaration.simpleName

getSuperclass	`// Should be able to do without resolution`
	`ksClassDeclaration.superTypes`
	`    .map { it.resolve() }`
	`    .filter { (it?.declaration as? KSClassDeclaration)?.classKind == ClassKind.CLASS }`

getTypeParameters	ksClassDeclaration.typeParameters

TypeParameterElement

Java	KSP equivalent
getBounds	ksTypeParameter.bounds
getEnclosingElement	ksTypeParameter.parentDeclaration
getGenericElement	ksTypeParameter.parentDeclaration

VariableElement

Java	KSP equivalent
getConstantValue	To be implemented
getEnclosingElement	ksValueParameter.parentDeclaration
getSimpleName	ksValueParameter.simpleName

ArrayType

Java	KSP equivalent
getComponentType	ksType.arguments.first()

DeclaredType

Java	KSP equivalent
asElement	ksType.declaration
getEnclosingType	ksType.declaration.parentDeclaration
getTypeArguments	ksType.arguments

ExecutableType

> A KSType for a function is just a signature represented by the FunctionN<R, T1, T2, ..., TN> family.

Java	KSP equivalent
getParameterTypes	ksType.declaration.typeParameters, ksFunctionDeclaration.parameters.map { it.type }
getReceiverType	ksFunctionDeclaration.parentDeclaration.asType(...)
getReturnType	ksType.declaration.typeParameters.last()
getThrownTypes	Not needed in Kotlin
getTypeVariables	ksFunctionDeclaration.typeParameters

IntersectionType

Java	KSP equivalent
getBounds	ksTypeParameter.bounds

TypeMirror

Java	KSP equivalent
getKind	Compare with types in KSBuiltIns for primitive types, Unit type, otherwise declared types

TypeVariable

Java	KSP equivalent
asElement	ksType.declaration
getLowerBound	To be decided. Only needed if capture is provided and explicit bound checking is needed.
getUpperBound	ksTypeParameter.bounds

WildcardType

Java	KSP equivalent
getExtendsBound	`if (ksTypeArgument.variance == Variance.COVARIANT) ksTypeArgument.type else null`
getSuperBound	`if (ksTypeArgument.variance == Variance.CONTRAVARIANT) ksTypeArgument.type else null`

Elements

Java	KSP equivalent
getAllAnnotationMirrors	KSDeclarations.annotations
getAllMembers	getAllFunctions, getAllProperties is to be implemented
getBinaryName	To be decided, see Java Specification
getConstantExpression	There is constant value, not expression
getDocComment	To be implemented
getElementValuesWithDefaults	To be implemented
getName	resolver.getKSNameFromString
getPackageElement	Package not supported, while package information can be retrieved. Operation on package is not possible for KSP
getPackageOf	Package not supported
getTypeElement	Resolver.getClassDeclarationByName

Java	KSP equivalent
hides	To be implemented

isDeprecated	`KsDeclaration.annotations.any {`
	`    it.annotationType.resolve()!!.declaration.qualifiedName!!.asString() ==`
	`Deprecated::class.qualifiedName`
	`}`

overrides	KSFunctionDeclaration.overrides / KSPropertyDeclaration.overrides (member function of respective class)
printElements	KSP has basic toString() implementation on most classes

Types

Java	KSP equivalent
asElement	ksType.declaration
asMemberOf	resolver.asMemberOf
boxedClass	Not needed
capture	To be decided
contains	KSType.isAssignableFrom
directSuperTypes	(ksType.declaration as KSClassDeclaration).superTypes
erasure	ksType.starProjection()
getArrayType	ksBuiltIns.arrayType.replace(...)
getDeclaredType	ksClassDeclaration.asType
getNoType	ksBuiltIns.nothingType / null
getNullType	Depending on the context, KSType.markNullable could be useful
getPrimitiveType	Not needed, check for KSBuiltins

Java	KSP equivalent
getWildcardType	Use Variance in places expecting KSTypeArgument
isAssignable	ksType.isAssignableFrom
isSameType	ksType.equals
isSubsignature	functionTypeA == functionTypeB / functionTypeA == functionTypeB.starProjection()
isSubtype	ksType.isAssignableFrom
unboxedType	Not needed

Incremental processing

Incremental processing is a processing technique that avoids re-processing of sources as much as possible. The primary goal of incremental processing is to reduce the turn-around time of a typical change-compile-test cycle. For general information, see Wikipedia's article on incremental computing.

To determine which sources are dirty (those that need to be reprocessed), KSP needs processors' help to identify which input sources correspond to which generated outputs. To help with this often cumbersome and error-prone process, KSP is designed to require only a minimal set of root sources that processors use as starting points to navigate the code structure. In other words, a processor needs to associate an output with the sources of the corresponding KSNode if the KSNode is obtained from any of the following:

- Resolver.getAllFiles
- Resolver.getSymbolsWithAnnotation
- Resolver.getClassDeclarationByName
- Resolver.getDeclarationsFromPackage

> Currently, only changes in Kotlin and Java sources are tracked. Changes to the classpath, namely to other modules or libraries, trigger a full re-processing of all sources by default. To track changes in classpath, set the Gradle property ksp.incremental.intermodule=true for an experimental implementation on JVM.

Incremental processing is currently enabled by default. To disable it, set the Gradle property ksp.incremental=false. To enable logs that dump the dirty set according to dependencies and outputs, use ksp.incremental.log=true. You can find these log files in the build output folder with a .log file extension.

Aggregating vs Isolating

Similar to the concepts in Gradle annotation processing, KSP supports both aggregating and isolating modes. Note that unlike Gradle annotation processing, KSP categorizes each output as either aggregating or isolating, rather than the entire processor.

An aggregating output can potentially be affected by any input changes, except removing files that don't affect other files. This means that any input change results in a rebuild of all aggregating outputs, which in turn means reprocessing of all corresponding registered, new, and modified source files.

As an example, an output that collects all symbols with a particular annotation is considered an aggregating output.

An isolating output depends only on its specified sources. Changes to other sources do not affect an isolating output. Note that unlike Gradle annotation processing, you can define multiple source files for a given output.

As an example, a generated class that is dedicated to an interface it implements is considered isolating.

To summarize, if an output might depend on new or any changed sources, it is considered aggregating. Otherwise, the output is isolating.

Here's a summary for readers familiar with Java annotation processing:

- In an isolating Java annotation processor, all the outputs are isolating in KSP.
- In an aggregating Java annotation processor, some outputs can be isolating and some can be aggregating in KSP.

How it is implemented

The dependencies are calculated by the association of input and output files, instead of annotations. This is a many-to-many relation.

The dirtiness propagation rules due to input-output associations are:

1. If an input file is changed, it will always be reprocessed.
2. If an input file is changed, and it is associated with an output, then all other input files associated with the same output will also be reprocessed. This is transitive, namely, invalidation happens repeatedly until there is no new dirty file.
3. All input files that are associated with one or more aggregating outputs will be reprocessed. In other words, if an input file isn't associated with any aggregating outputs, it won't be reprocessed (unless it meets 1. or 2. in the above).

Reasons are:

1. If an input is changed, new information can be introduced and therefore processors need to run again with the input.
2. An output is made out of a set of inputs. Processors may need all the inputs to regenerate the output.
3. aggregating=true means that an output may potentially depend on new information, which can come from either new files, or changed, existing files. aggregating=false means that processor is sure that the information only comes from certain input files and never from other or new files.

Example 1

A processor generates outputForA after reading class A in A.kt and class B in B.kt, where A extends B. The processor got A by Resolver.getSymbolsWithAnnotation and then got B by KSClassDeclaration.superTypes from A. Because the inclusion of B is due to A, B.kt doesn't need to be specified in dependencies for outputForA. You can still specify B.kt in this case, but it is unnecessary.

```
// A.kt
@Interesting
class A : B()

// B.kt
open class B

// Example1Processor.kt
class Example1Processor : SymbolProcessor {
    override fun process(resolver: Resolver) {
        val declA = resolver.getSymbolsWithAnnotation("Interesting").first() as KSClassDeclaration
        val declB = declA.superTypes.first().resolve().declaration
        // B.kt isn't required, because it can be deduced as a dependency by KSP
        val dependencies = Dependencies(aggregating = true, declA.containingFile!!)
        // outputForA.kt
        val outputName = "outputFor${declA.simpleName.asString()}"
        // outputForA depends on A.kt and B.kt
        val output = codeGenerator.createNewFile(dependencies, "com.example", outputName, "kt")
        output.write("// $declA : $declB\n".toByteArray())
        output.close()
    }
    // ...
}
```

Example 2

Consider that a processor generates outputA after reading sourceA and outputB after reading sourceB.

When sourceA is changed:

- If outputB is aggregating, both sourceA and sourceB are reprocessed.
- If outputB is isolating, only sourceA is reprocessed.

When sourceC is added:

- If outputB is aggregating, both sourceC and sourceB are reprocessed.
- If outputB is isolating, only sourceC is reprocessed.

When sourceA is removed, nothing needs to be reprocessed.

When sourceB is removed, nothing needs to be reprocessed.

How file dirtiness is determined

A dirty file is either directly changed by users or indirectly affected by other dirty files. KSP propagates dirtiness in two steps:

- Propagation by resolution tracing: Resolving a type reference (implicitly or explicitly) is the only way to navigate from one file to another. When a type reference is resolved by a processor, a changed or affected file that contains a change that may potentially affect the resolution result will affect the file containing that reference.
- Propagation by input-output correspondence: If a source file is changed or affected, all other source files having some output in common with that file are affected.

Note that both of them are transitive and the second forms equivalence classes.

Reporting bugs

To report a bug, please set Gradle properties ksp.incremental=true and ksp.incremental.log=true, and perform a clean build. This build produces two log files:

- build/kspCaches/<source set>/logs/kspDirtySet.log
- build/kspCaches/<source set>/logs/kspSourceToOutputs.log

You can then run successive incremental builds, which will generate two additional log files:

- build/kspCaches/<source set>/logs/kspDirtySetByDeps.log
- build/kspCaches/<source set>/logs/kspDirtySetByOutputs.log

These logs contain file names of sources and outputs, plus the timestamps of the builds.

Multiple round processing

KSP supports multiple round processing, or processing files over multiple rounds. It means that subsequent rounds use an output from previous rounds as additional input.

Changes to your processor

To use multiple round processing, the SymbolProcessor.process() function needs to return a list of deferred symbols (List<KSAnnotated>) for invalid symbols. Use KSAnnotated.validate() to filter invalid symbols to be deferred to the next round.

The following sample code shows how to defer invalid symbols by using a validation check:

```
override fun process(resolver: Resolver): List<KSAnnotated> {
    val symbols = resolver.getSymbolsWithAnnotation("com.example.annotation.Builder")
    val result = symbols.filter { !it.validate() }
    symbols
        .filter { it is KSClassDeclaration && it.validate() }
        .map { it.accept(BuilderVisitor(), Unit) }
    return result
}
```

Multiple round behavior

Deferring symbols to the next round

Processors can defer the processing of certain symbols to the next round. When a symbol is deferred, processor is waiting for other processors to provide additional information. It can continue deferring the symbol as many rounds as needed. Once the other processors provide the required information, the processor can then process the deferred symbol. Processor should only defer invalid symbols which are lacking necessary information. Therefore, processors should not defer symbols from classpath, KSP will also filter out any deferred symbols that are not from source code.

As an example, a processor that creates a builder for an annotated class might require all parameter types of its constructors to be valid (resolved to a concrete type). In the first round, one of the parameter type is not resolvable. Then in the second round, it becomes resolvable because of the generated files from the first round.

Validating symbols

A convenient way to decide if a symbol should be deferred is through validation. A processor should know which information is necessary to properly process the symbol. Note that validation usually requires resolution which can be expensive, so we recommend checking only what is required. Continuing with the previous example, an ideal validation for the builder processor checks only whether all resolved parameter types of the constructors of annotated symbols contain isError == false.

KSP provides a default validation utility. For more information, see the Advanced section.

Termination condition

Multiple round processing terminates when a full round of processing generates no new files. If unprocessed deferred symbols still exist when the termination condition is met, KSP logs an error message for each processor with unprocessed deferred symbols.

Files accessible at each round

Both newly generated files and existing files are accessible through a Resolver. KSP provides two APIs for accessing files: Resolver.getAllFiles() and Resolver.getNewFiles(). getAllFiles() returns a combined list of both existing files and newly generated files, while getNewFiles() returns only newly generated files.

Changes to getSymbolsAnnotatedWith()

To avoid unnecessary reprocessing of symbols, getSymbolsAnnotatedWith() returns only those symbols found in newly generated files, together with the symbols from deferred symbols from the last round.

Processor instantiating

A processor instance is created only once, which means you can store information in the processor object to be used for later rounds.

Information consistent cross rounds

All KSP symbols will not be reusable across multiple rounds, as the resolution result can potentially change based on what was generated in a previous round. However, since KSP does not allow modifying existing code, some information such as the string value for a symbol name should still be reusable. To summarize, processors can store information from previous rounds but need to bear in mind that this information might be invalid in future rounds.

Error and exception handling

When an error (defined by processor calling KSPLogger.error()) or exception occurs, processing stops after the current round completes. All processors will call the onError() method and will not call the finish() method.

Note that even though an error has occurred, other processors continue processing normally for that round. This means that error handling occurs after processing has completed for the round.

Upon exceptions, KSP will try to distinguish the exceptions from KSP and exceptions from processors. Exceptions will result in a termination of processing immediately and be logged as an error in KSPLogger. Exceptions from KSP should be reported to KSP developers for further investigation. At the end of the round where exceptions or errors happened, all processors will invoke onError() function to do their own error handling.

KSP provides a default no-op implementation for onError() as part of the SymbolProcessor interface. You can override this method to provide your own error handling logic.

Advanced

Default behavior for validation

The default validation logic provided by KSP validates all directly reachable symbols inside the enclosing scope of the symbol that is being validated. Default validation checks whether references in the enclosed scope are resolvable to a concrete type but does not recursively dive into the referenced types to perform validation.

Write your own validation logic

Default validation behavior might not be suitable for all cases. You can reference KSValidateVisitor and write your own validation logic by providing a custom predicate lambda, which is then used by KSValidateVisitor to filter out symbols that need to be checked.

KSP with Kotlin Multiplatform

For a quick start, see a sample Kotlin Multiplatform project defining a KSP processor.

Starting from KSP 1.0.1, applying KSP on a multiplatform project is similar to that on a single platform, JVM project. The main difference is that, instead of writing the ksp(...) configuration in dependencies, add(ksp<Target>) or add(ksp<SourceSet>) is used to specify which compilation targets need symbol processing, before compilation.

```
plugins {
    kotlin("multiplatform")
    id("com.google.devtools.ksp")
}

kotlin {
    jvm {
        withJava()
    }
    linuxX64() {
        binaries {
            executable()
        }
    }
    sourceSets {
        val commonMain by getting
        val linuxX64Main by getting
        val linuxX64Test by getting
    }
}

dependencies {
    add("kspCommonMainMetadata", project(":test-processor"))
    add("kspJvm", project(":test-processor"))
    add("kspJvmTest", project(":test-processor")) // Not doing anything because there's no test source set for JVM
    // There is no processing for the Linux x64 main source set, because kspLinuxX64 isn't specified
    add("kspLinuxX64Test", project(":test-processor"))
}
```

Compilation and processing

In a multiplatform project, Kotlin compilation may happen multiple times (main, test, or other build flavors) for each platform. So is symbol processing. A symbol processing task is created whenever there is a Kotlin compilation task and a corresponding ksp<Target> or ksp<SourceSet> configuration is specified.

For example, in the above build.gradle.kts, there are 4 compilation tasks: common/metadata, JVM main, Linux x64 main, Linux x64 test, and 3 symbol processing tasks: common/metadata, JVM main, Linux x64 test.

Avoid the ksp(...) configuration on KSP 1.0.1+

Before KSP 1.0.1, there is only one, unified ksp(...) configuration available. Therefore, processors either applies to all compilation targets, or nothing at all. Note that the ksp(...) configuration not only applies to the main source set, but also the test source set if it exists, even on traditional, non-multiplatform projects. This brought unnecessary overheads to build time.

Starting from KSP 1.0.1, per-target configurations are provided as shown in the above example. In the future:

1. For multiplatform projects, the ksp(...) configuration will be deprecated and removed.
2. For single platform projects, the ksp(...) configuration will only apply to the main, default compilation. Other targets like test will need to specify kspTest(...) in order to apply processors.

Starting from KSP 1.0.1, there is an early access flag -DallowAllTargetConfiguration=false to switch to the more efficient behavior. If the current behavior is causing performance issues, please give it a try. The default value of the flag will be flipped from true to false on KSP 2.0.

Running KSP from command line

KSP is a Kotlin compiler plugin and needs to run with Kotlin compiler. Download and extract them.

```
#!/bin/bash

# Kotlin compiler
wget https://github.com/JetBrains/kotlin/releases/download/v1.9.0/kotlin-compiler-1.9.0.zip
unzip kotlin-compiler-1.9.0.zip

# KSP
wget https://github.com/google/ksp/releases/download/1.9.0-1.0.11/artifacts.zip
unzip artifacts.zip
```

To run KSP with kotlinc, pass the -Xplugin option to kotlinc.

-Xplugin=/path/to/symbol-processing-cmdline-1.9.0-1.0.11.jar

This is different from the symbol-processing-1.9.0-1.0.11.jar, which is designed to be used with kotlin-compiler-embeddable when running with Gradle. The command line kotlinc needs symbol-processing-cmdline-1.9.0-1.0.11.jar.

You'll also need the API jar.

-Xplugin=/path/to/symbol-processing-api-1.9.0-1.0.11.jar

See the complete example:

```
#!/bin/bash

KSP_PLUGIN_ID=com.google.devtools.ksp.symbol-processing
KSP_PLUGIN_OPT=plugin:$KSP_PLUGIN_ID

KSP_PLUGIN_JAR=./com/google/devtools/ksp/symbol-processing-cmdline/1.9.0-1.0.11/symbol-processing-cmdline-1.9.0-1.0.11.jar
KSP_API_JAR=./com/google/devtools/ksp/symbol-processing-api/1.9.0-1.0.11/symbol-processing-api-1.9.0-1.0.11.jar
KOTLINC=./kotlinc/bin/kotlinc

AP=/path/to/your-processor.jar

mkdir out
$KOTLINC \
        -Xplugin=$KSP_PLUGIN_JAR \
        -Xplugin=$KSP_API_JAR \
        -Xallow-no-source-files \
        -P $KSP_PLUGIN_OPT:apclasspath=$AP \
        -P $KSP_PLUGIN_OPT:projectBaseDir=. \
        -P $KSP_PLUGIN_OPT:classOutputDir=./out \
        -P $KSP_PLUGIN_OPT:javaOutputDir=./out \
        -P $KSP_PLUGIN_OPT:kotlinOutputDir=./out \
        -P $KSP_PLUGIN_OPT:resourceOutputDir=./out \
        -P $KSP_PLUGIN_OPT:kspOutputDir=./out \
        -P $KSP_PLUGIN_OPT:cachesDir=./out \
        -P $KSP_PLUGIN_OPT:incremental=false \
        -P $KSP_PLUGIN_OPT:apoption=key1=value1 \
        -P $KSP_PLUGIN_OPT:apoption=key2=value2 \
        $*
```

KSP FAQ

Why KSP?

KSP has several advantages over kapt:

- It is faster.
- The API is more fluent for Kotlin users.
- It supports multiple round processing on generated Kotlin sources.
- It is being designed with multiplatform compatibility in mind.

Why is KSP faster than kapt?

kapt has to parse and resolve every type reference in order to generate Java stubs, whereas KSP resolves references on-demand. Delegating to javac also takes time.

Additionally, KSP's incremental processing model has a finer granularity than just isolating and aggregating. It finds more opportunities to avoid reprocessing everything. Also, because KSP traces symbol resolutions dynamically, a change in a file is less likely to pollute other files and therefore the set of files to be reprocessed is smaller. This is not possible for kapt because it delegates processing to javac.

Is KSP Kotlin-specific?

KSP can process Java sources as well. The API is unified, meaning that when you parse a Java class and a Kotlin class you get a unified data structure in KSP.

How to upgrade KSP?

KSP has API and implementation. The API rarely changes and is backward compatible: there can be new interfaces, but old interfaces never change. The implementation is tied to a specific compiler version. With the new release, the supported compiler version can change.

Processors only depend on API and therefore are not tied to compiler versions. However, users of processors need to bump KSP version when bumping the compiler version in their project. Otherwise, the following error will occur:

```
ksp-a.b.c is too old for kotlin-x.y.z. Please upgrade ksp or downgrade kotlin-gradle-plugin
```

> Users of processors don't need to bump processor's version because processors only depend on API.

For example, some processor is released and tested with KSP 1.0.1, which depends strictly on Kotlin 1.6.0. To make it work with Kotlin 1.6.20, the only thing you need to do is bump KSP to a version (for example, KSP 1.1.0) that is built for Kotlin 1.6.20.

Can I use a newer KSP implementation with an older Kotlin compiler?

If the language version is the same, Kotlin compiler is supposed to be backward compatible. Bumping Kotlin compiler should be trivial most of the time. If you need a newer KSP implementation, please upgrade the Kotlin compiler accordingly.

How often do you update KSP?

KSP tries to follow Semantic Versioning as close as possible. With KSP version major.minor.patch,

- major is reserved for incompatible API changes. There is no pre-determined schedule for this.
- minor is reserved for new features. This is going to be updated approximately quarterly.
- patch is reserved for bug fixes and new Kotlin releases. It's updated roughly monthly.

Usually a corresponding KSP release is available within a couple of days after a new Kotlin version is released, including the pre-releases (Beta or RC).

Besides Kotlin, are there other version requirements to libraries?

Here is a list of requirements for libraries/infrastructures:

- Android Gradle Plugin 4.1.0+
- Gradle 6.5+

What is KSP's future roadmap?

The following items have been planned:

- Support new Kotlin compiler
- Improve support to multiplatform. For example, running KSP on a subset of targets/sharing computations between targets.
- Improve performance. There are a bunch of optimizations to be done!
- Keep fixing bugs.

Please feel free to reach out to us in the #ksp channel in Kotlin Slack (get an invite) if you would like to discuss any ideas. Filing GitHub issues/feature requests or pull requests are also welcome!

Learning materials overview

You can use the following materials and resources for learning Kotlin:

- Basic syntax – get a quick overview of the Kotlin syntax.
- Idioms – learn how to write idiomatic Kotlin code for popular cases.
  - Java to Kotlin migration guide: Strings – learn how to perform typical tasks with strings in Java and Kotlin.
  - Java to Kotlin migration guide: Collections — learn how to perform typical tasks with collections in Java and Kotlin.
  - Java to Kotlin migration guide: Nullability — learn how to handle nullability in Java and Kotlin.
- Kotlin Koans – complete exercises to learn the Kotlin syntax. Each exercise is created as a failing unit test and your job is to make it pass. Recommended for developers with Java experience.
- Kotlin by example – review a set of small and simple annotated examples for the Kotlin syntax.
- Kotlin Core track by JetBrains Academy – learn all the Kotlin essentials while creating working applications step by step.
- Kotlin books – find books we've reviewed and recommend for learning Kotlin.
- Kotlin tips – watch short videos where the Kotlin team shows you how to use Kotlin in a more efficient and idiomatic way, so you can have more fun when writing code.
- Advent of Code puzzles – learn idiomatic Kotlin and test your language skills by completing short and fun tasks.
- Kotlin hands-on tutorials – complete long-form tutorials to fully grasp a technology. These tutorials guide you through a self-contained project related to a specific topic.
- Kotlin for Java Developers – learn the similarities and differences between Java and Kotlin in this course on Coursera.
- Kotlin documentation in PDF format – read our documentation offline.

Kotlin Koans

Kotlin Koans are a series of exercises designed primarily for Java developers, to help you become familiar with the Kotlin syntax. Each exercise is created as a failing unit test, and your job is to make it pass. You can complete the Kotlin Koans tasks in one of the following ways:

- You can play with Koans online.
- You can perform the tasks right inside IntelliJ IDEA or Android Studio by installing the JetBrains Academy plugin and choosing the Kotlin Koans course.

Whatever way you choose to solve koans, you can see the solution for each task:

- In the online version, click Show answer.
- For the JetBrains Academy plugin, try to complete the task first and then choose Peek solution if your answer is incorrect.

We recommend you check the solution after implementing the task to compare your answer with the proposed one. Make sure you don't cheat!

Kotlin tips

Kotlin Tips is a series of short videos where members of the Kotlin team show how to use Kotlin in a more efficient and idiomatic way to have more fun when writing code.

Subscribe to our YouTube channel to not miss new Kotlin Tips videos.

null + null in Kotlin

What happens when you add null + null in Kotlin, and what does it return? Sebastian Aigner addresses this mystery in our latest quick tip. Along the way, he also shows why there's no reason to be scared of nullables:

Watch video online.

Deduplicating collection items

Got a Kotlin collection that contains duplicates? Need a collection with only unique items? Let Sebastian Aigner show you how to remove duplicates from your lists, or turn them into sets in this Kotlin tip:

Watch video online.

The suspend and inline mystery

How come functions like repeat(), map() and filter() accept suspending functions in their lambdas, even though their signatures aren't coroutines-aware? In this episode of Kotlin Tips Sebastian Aigner solves the riddle: it has something to do with the inline modifier:

Watch video online.

Unshadowing declarations with their fully qualified name

Shadowing means having two declarations in a scope have the same name. So, how do you pick? In this episode of Kotlin Tips Sebastian Aigner shows you a simple Kotlin trick to call exactly the function that you need, using the power of fully qualified names:

Watch video online.

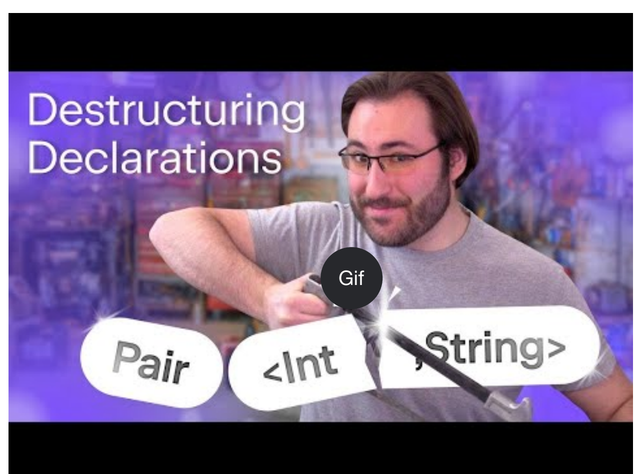
Return and throw with the Elvis operator

Elvis has entered the building once more! Sebastian Aigner explains why the operator is named after the famous singer, and how you can use ?: in Kotlin to return or throw. The magic behind the scenes? The Nothing type.

Watch video online.

Destructuring declarations

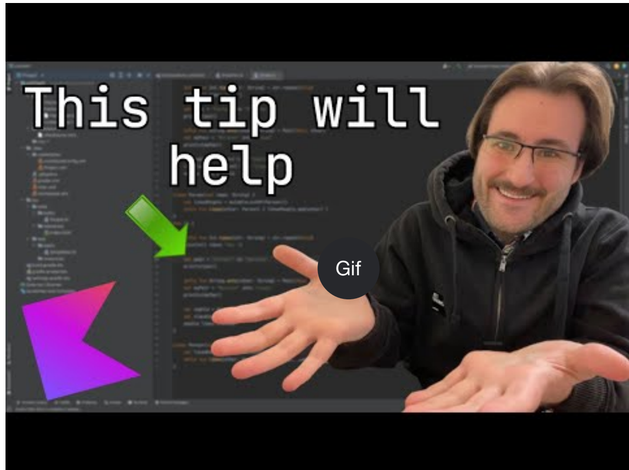
With destructuring declarations in Kotlin, you can create multiple variables from a single object, all at once. In this video Sebastian Aigner shows you a selection of things that can be destructured – pairs, lists, maps, and more. And what about your own objects? Kotlin's component functions provide an answer for those as well:



Watch video online.

Operator functions with nullable values

In Kotlin, you can override operators like addition and subtraction for your classes and supply your own logic. But what if you want to allow null values, both on their left and right sides? In this video, Sebastian Aigner answers this question:



Watch video online.

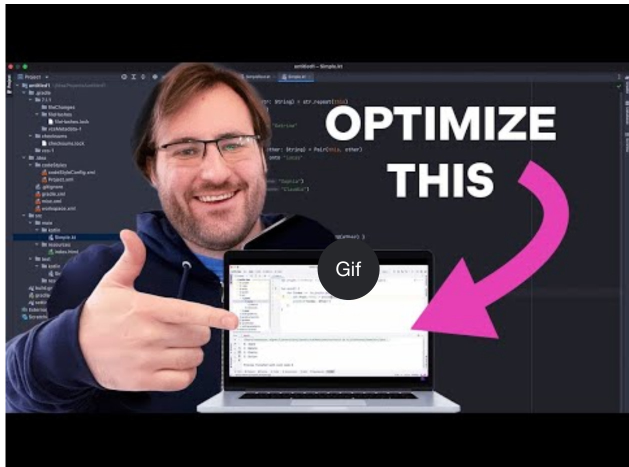
Timing code

Watch Sebastian Aigner give a quick overview of the measureTimedValue() function, and learn how you can time your code:

Watch video online.

Improving loops

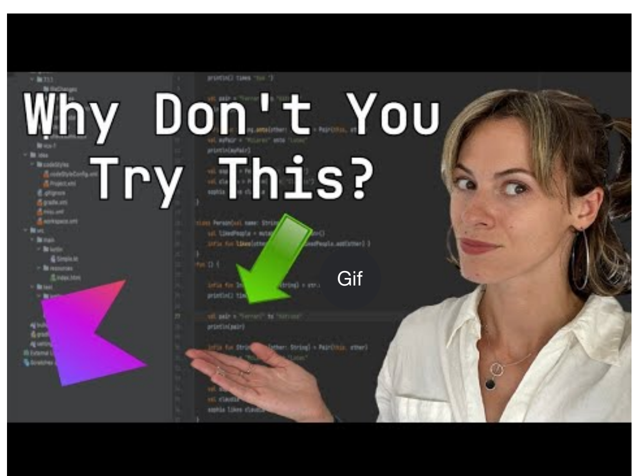
In this video, Sebastian Aigner will demonstrate how to improve loops to make your code more readable, understandable, and concise:



Watch video online.

Strings

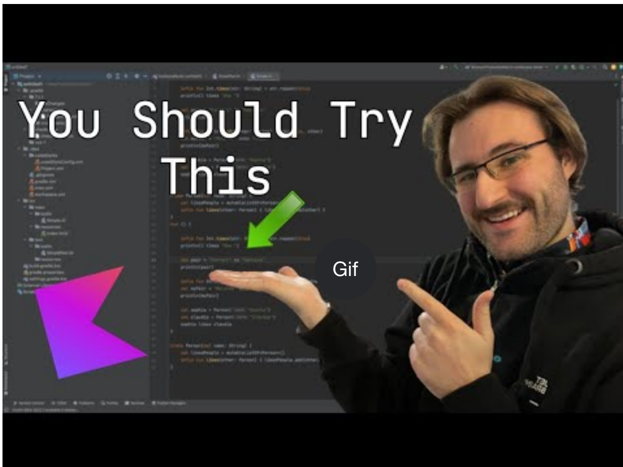
In this episode, Kate Petrova shows three tips to help you work with Strings in Kotlin:



Watch video online.

Doing more with the Elvis operator

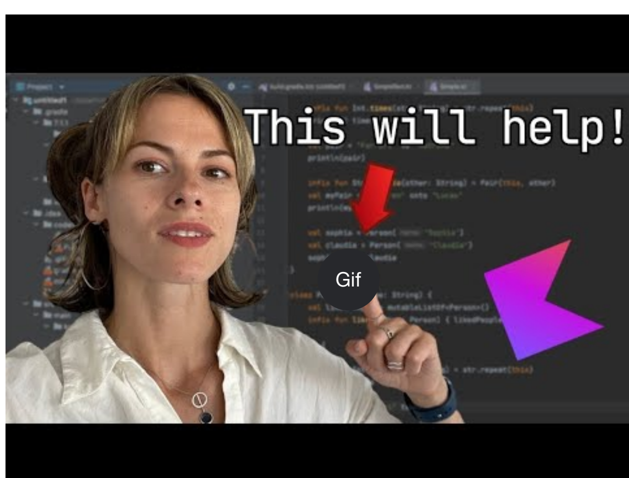
In this video, Sebastian Aigner will show how to add more logic to the Elvis operator, such as logging to the right part of the operator:



Watch video online.

Kotlin collections

In this episode, Kate Petrova shows three tips to help you work with Kotlin Collections:



Watch video online.

What's next?

- See the complete list of Kotlin Tips in our YouTube playlist
- Learn how to write idiomatic Kotlin code for popular cases

Kotlin books

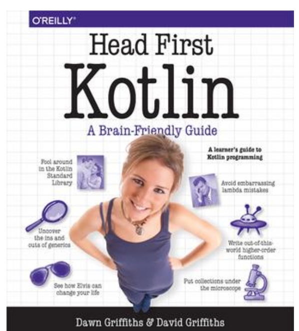
More and more authors write books for learning Kotlin in different languages. We are very thankful to all of them and appreciate all their efforts in helping us increase a number of professional Kotlin developers.

Here are just a few books we've reviewed and recommend you for learning Kotlin. You can find more books on our community website.

Atomic Kotlin

Atomic Kotlin is for both beginning and experienced programmers!

From Bruce Eckel, author of the multi-award-winning Thinking in C++ and Thinking in Java, and Svetlana Isakova, Kotlin Developer Advocate at JetBrains, comes a book that breaks the language concepts into small, easy-to-digest "atoms", along with a free course consisting of exercises supported by hints and solutions directly inside IntelliJ IDEA!

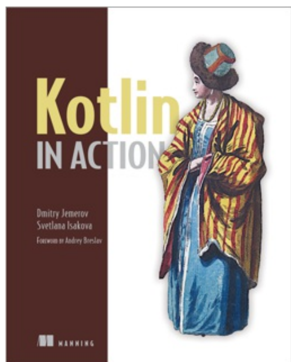


Head First Kotlin

Head First Kotlin is a complete introduction to coding in Kotlin. This hands-on book helps you learn the Kotlin language with a unique method that goes beyond syntax and how-to manuals and teaches you how to think like a great Kotlin developer.

You'll learn everything from language fundamentals to collections, generics, lambdas, and higher-order functions. Along the way, you'll get to play with both object-oriented and functional programming.

If you want to really understand Kotlin, this is the book for you.

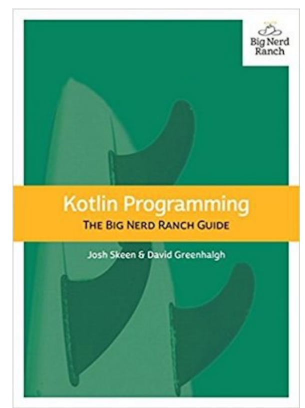


Kotlin in Action

Kotlin in Action teaches you to use the Kotlin language for production-quality applications. Written for experienced Java developers, this example-rich book goes further than most language books, covering interesting topics like building DSLs with natural language syntax.

The book is written by Dmitry Jemerov and Svetlana Isakova, developers on the Kotlin team.

Chapter 6, covering the Kotlin type system, and chapter 11, covering DSLs, are available as a free preview on the publisher web site.



Kotlin Programming: The Big Nerd Ranch Guide

Kotlin Programming: The Big Nerd Ranch Guide

In this book you will learn to work effectively with the Kotlin language through carefully considered examples designed to teach you Kotlin's elegant style and features.

Starting from first principles, you will work your way to advanced usage of Kotlin, empowering you to create programs that are more reliable with less code.

Programming Kotlin

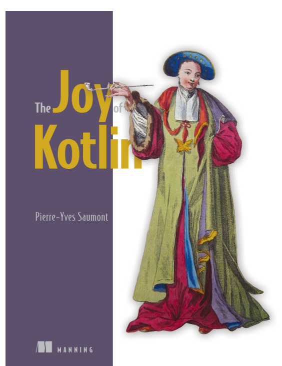
Programming Kotlin is written by Venkat Subramaniam.

Programmers don't just use Kotlin, they love it. Even Google has adopted it as a first-class language for Android development.

With Kotlin, you can intermix imperative, functional, and object-oriented styles of programming and benefit from the approach that's most suitable for the problem at hand.

Learn to use the many features of this highly concise, fluent, elegant, and expressive statically typed language with easy-to-understand examples.

Learn to write maintainable, high-performing JVM and Android applications, create DSLs, program asynchronously, and much more.



The Joy of Kotlin

The Joy of Kotlin teaches you the right way to code in Kotlin.

In this insight-rich book, you'll master the Kotlin language while exploring coding techniques that will make you a better developer no matter what language you use. Kotlin natively supports a functional style of programming, so seasoned author Pierre-Yves Saumont begins by reviewing the FP principles of immutability, referential transparency, and the separation between functions and effects.

Then, you'll move deeper into using Kotlin in the real world, as you learn to handle errors and data properly, encapsulate shared state mutations, and work with laziness.

This book will change the way you code — and give you back some of the joy you had when you first started.

Advent of Code puzzles in idiomatic Kotlin

Advent of Code is an annual December event, where holiday-themed puzzles are published every day from December 1 to December 25. With the permission of Eric Wastl, creator of Advent of Code, we'll show how to solve these puzzles using the idiomatic Kotlin style:

- Advent of Code 2021
- Advent of Code 2020

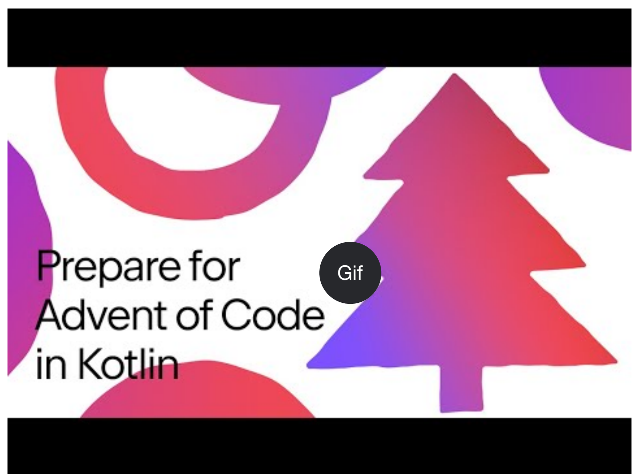
Advent of Code 2021

- Get ready
- Day 1: Sonar sweep
- Day 2: Dive!
- Day 3: Binary diagnostic
- Day 4: Giant squid

Get ready

We'll take you through the basic tips on how to get up and running with solving Advent of Code challenges with Kotlin:

- Read our blog post about Advent of Code 2021
- Use this GitHub template to create projects
- Check out the welcome video by Kotlin Developer Advocate, Sebastian Aigner:

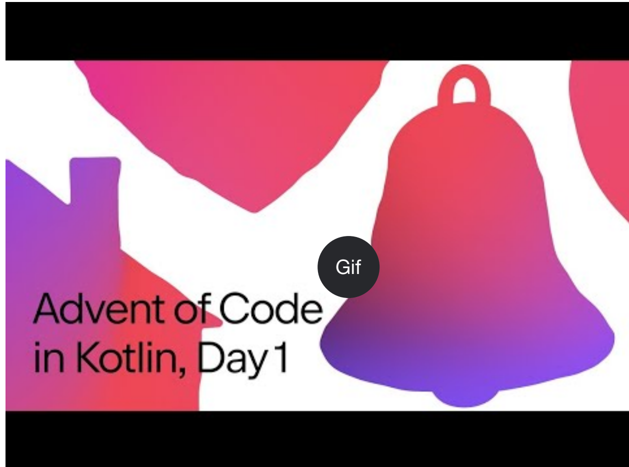


Watch video online.

Day 1: Sonar sweep

Apply windowed and count functions to work with pairs and triplets of integers.

- Read the puzzle description on Advent of Code
- Check out the solution from Anton Arhipov on the Kotlin Blog or watch the video:

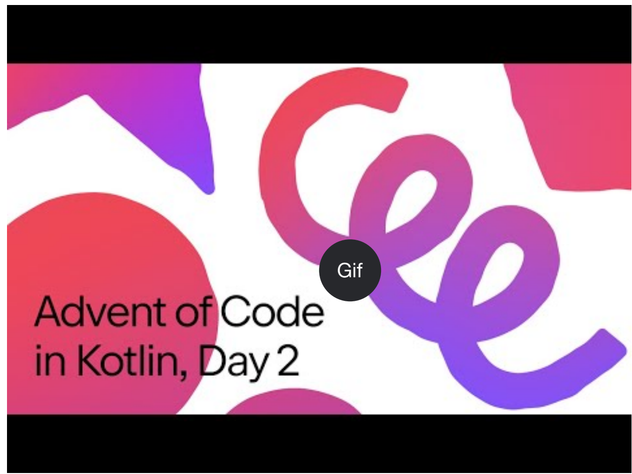


Watch video online.

Day 2: Dive!

Learn about destructuring declarations and the when expression.

- Read the puzzle description on Advent of Code
- Check out the solution from Pasha Finkelshteyn on GitHub or watch the video:



Watch video online.

Day 3: Binary diagnostic

Explore different ways to work with binary numbers.

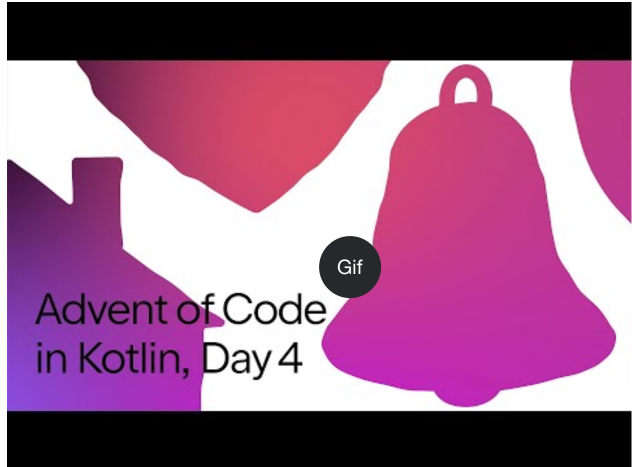
- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on Kotlin Blog or watch the video:

Watch video online.

Day 4: Giant squid

Learn how to parse the input and introduce some domain classes for more convenient processing.

- Read the puzzle description on Advent of Code
- Check out the solution from Anton Arhipov on the GitHub or watch the video:



Watch video online.

Advent of Code 2020

> You can find all the solutions for the Advent of Code 2020 puzzles in our GitHub repository.

- Day 1: Report repair
- Day 2: Password philosophy
- Day 3: Toboggan trajectory
- Day 4: Passport processing
- Day 5: Binary boarding
- Day 6: Custom customs

* Day 7: Handy haversacks
* Day 8: Handheld halting
* Day 9: Encoding error

Day 1: Report repair

Explore input handling, iterating over a list, different ways of building a map, and using the let function to simplify your code.

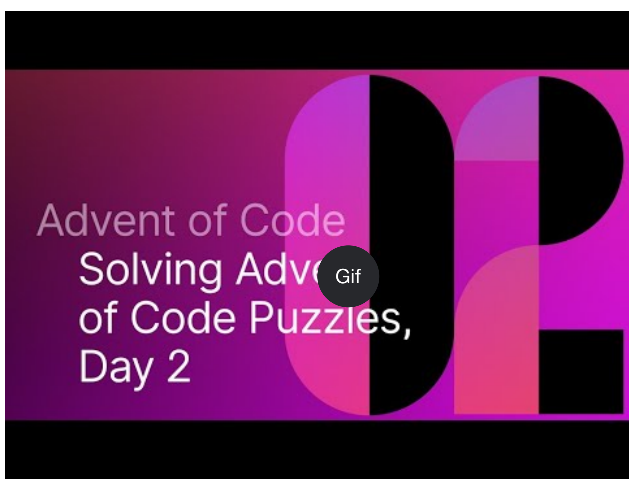
* Read the puzzle description on Advent of Code
* Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

Watch video online.

Day 2: Password philosophy

Explore string utility functions, regular expressions, operations on collections, and how the let function can be helpful to transform your expressions.

* Read the puzzle description on Advent of Code
* Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

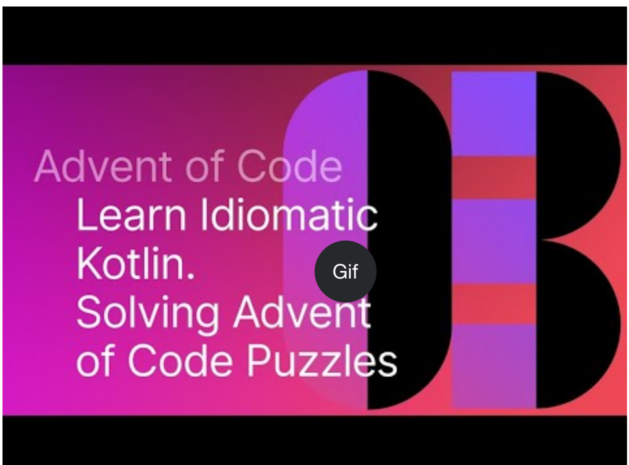


Watch video online.

Day 3: Toboggan trajectory

Compare imperative and more functional code styles, work with pairs and the reduce() function, edit code in the column selection mode, and fix integer overflows.

- Read the puzzle description on Advent of Code
- Check out the solution from Mikhail Dvorkin on GitHub or watch the video:

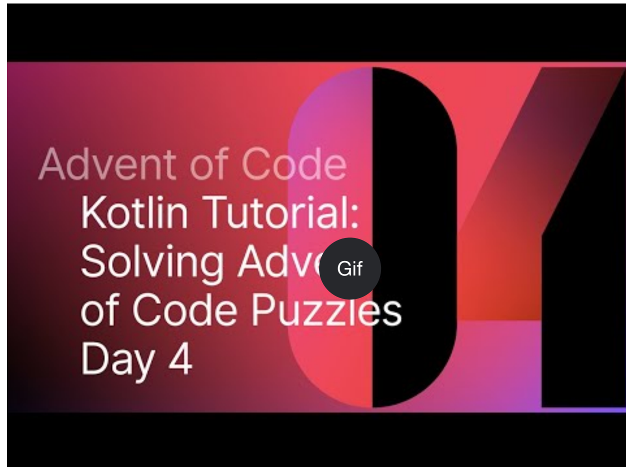


Watch video online.

Day 4: Passport processing

Apply the when expression and explore different ways of how to validate the input: utility functions, working with ranges, checking set membership, and matching a particular regular expression.

- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on the Kotlin Blog or watch the video:

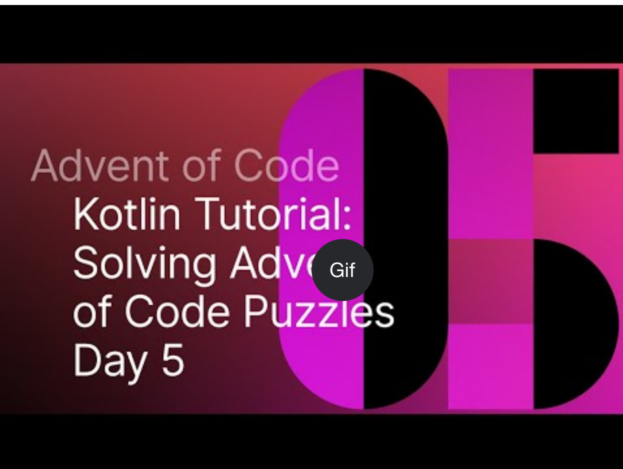


Watch video online.

Day 5: Binary boarding

Use the Kotlin standard library functions (replace(), toInt(), find()) to work with the binary representation of numbers, explore powerful local functions, and learn how to use the max() function in Kotlin 1.5.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

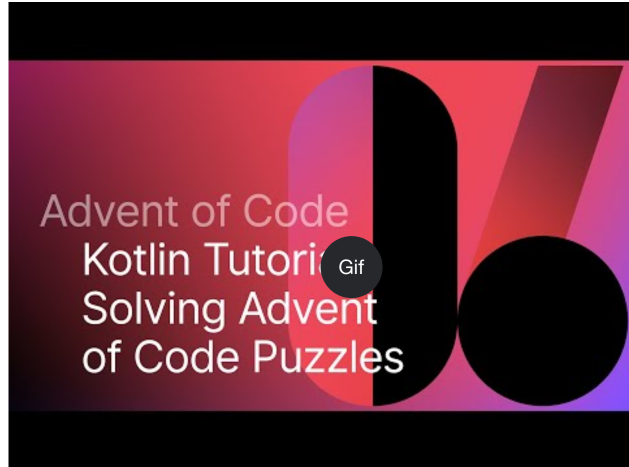


Watch video online.

Day 6: Custom customs

Learn how to group and count characters in strings and collections using the standard library functions: map(), reduce(), sumOf(), intersect(), and union().

- Read the puzzle description on Advent of Code
- Check out the solution from Anton Arhipov on the Kotlin Blog or watch the video:



Watch video online.

Day 7: Handy haversacks

Learn how to use regular expressions, use Java's compute() method for HashMaps from Kotlin for dynamic calculations of the value in the map, use the forEachLine() function to read files, and compare two types of search algorithms: depth-first and breadth-first.

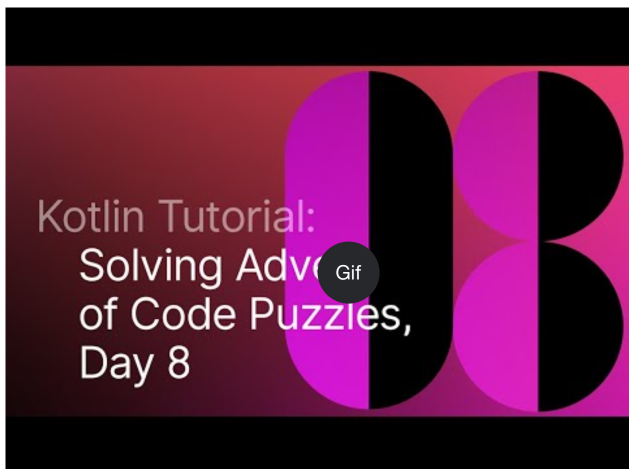
- Read the puzzle description on Advent of Code
- Check out the solution from Pasha Finkelshteyn on the Kotlin Blog or watch the video:

Watch video online.

Day 8: Handheld halting

Apply sealed classes and lambdas to represent instructions, apply Kotlin sets to discover loops in the program execution, use sequences and the sequence { } builder function to construct a lazy collection, and try the experimental measureTimedValue() function to check performance metrics.

- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on the Kotlin Blog or watch the video:

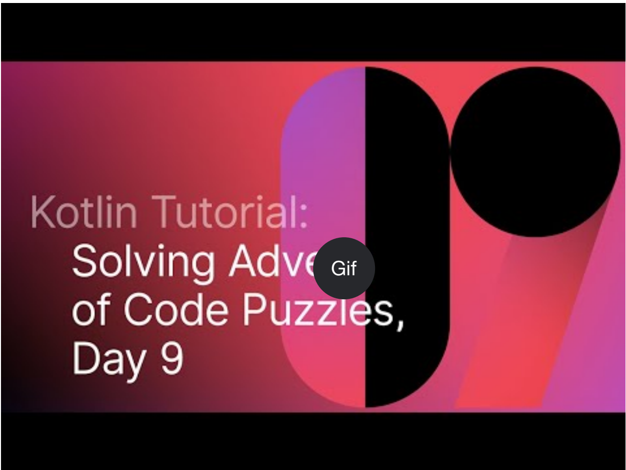


Watch video online.

Day 9: Encoding error

Explore different ways to manipulate lists in Kotlin using the any(), firstOrNull(), firstNotNullOfOrNull(), windowed(), takeIf(), and scan() functions, which exemplify an idiomatic Kotlin style.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:



Watch video online.

What's next?

* Complete more tasks with Kotlin Koans
* Create working applications with the free Kotlin Core track by JetBrains Academy

Learning Kotlin with JetBrains Academy plugin

With the JetBrains Academy plugin, available both in Android Studio and IntelliJ IDEA, you can learn Kotlin through code practicing tasks.

Take a look at the Learner Start Guide, which will get you started with the Kotlin Koans course inside IntelliJ IDEA. Solve interactive coding challenges and get instant feedback right inside the IDE.

If you want to use the JetBrains Academy plugin for teaching, read Teaching Kotlin with JetBrains Academy plugin.

Teaching Kotlin with JetBrains Academy plugin

With the JetBrains Academy plugin, available both in Android Studio and IntelliJ IDEA, you can teach Kotlin through code practicing tasks. Take a look at the Educator Start Guide to learn how to create a simple Kotlin course that includes a set of programming tasks and integrated tests.

If you want to use the JetBrains Academy plugin to learn Kotlin, read Learning Kotlin with JetBrains Academy plugin.

Participate in the Kotlin Early Access Preview

You can participate in the Kotlin Early Access Preview (EAP) to try out the latest Kotlin features before they are released.

We ship a few Beta (Beta) and Release Candidate (RC) builds before every feature (1.x) and incremental (1.x.y) release.

We'll be very thankful if you find and report bugs to our issue tracker YouTrack. It is very likely that we'll be able to fix them before the final release, which means you won't need to wait until the next Kotlin release for your issues to be addressed.

By participating in the Early Access Preview and reporting bugs, you contribute to Kotlin and help us make it better for everyone in the growing Kotlin community. We appreciate your help a lot!

If you have any questions and want to participate in discussions, you are welcome to join the #eap channel in Kotlin Slack. In this channel, you can also get notifications about new EAP builds.

Install the Kotlin EAP Plugin for IDEA or Android Studio

> By participating in the EAP, you expressly acknowledge that the EAP version may not be reliable, may not work as intended, and may contain errors.
>
> Please note that we don't provide any guarantees of compatibility between EAP and final versions of the same release.

If you have already installed the EAP version and want to work on projects that were created previously, check our instructions on how to configure your build to support this version.

How the EAP can help you be more productive with Kotlin

- Prepare for the Stable release. If you work on a complex multimodule project, participating in the EAP may streamline your experience when you adopt the Stable release version. The sooner you update to the Stable version, the sooner you can take advantage of its performance improvements and new language features.

  The migration of huge and complex projects might take a while, not only because of their size, but also because some specific use cases may not have been covered by the Kotlin team yet. By participating in the EAP and continuously testing new versions of Kotlin, you can provide us with early feedback about your specific use cases. This will help us address as many issues as possible and ensure you can safely update to the Stable version when it's released. Check out how Slack benefits from testing Android, Kotlin, and Gradle pre-release versions.

- Keep your library up-to-date. If you're a library author, updating to the new Kotlin version is extremely important. Using older versions could block your users from updating Kotlin in their projects. Working with EAP versions allows you to support the latest Kotlin versions in your library almost immediately with the Stable release, which makes your users happier and your library more popular.

- Share the experience. If you're a Kotlin enthusiast and enjoy contributing to the Kotlin ecosystem by creating educational content, trying new features in the Kotlin EAP allows you to be among the first to share the experience of using the new cool features with the community.

Build details

No preview versions are currently available.

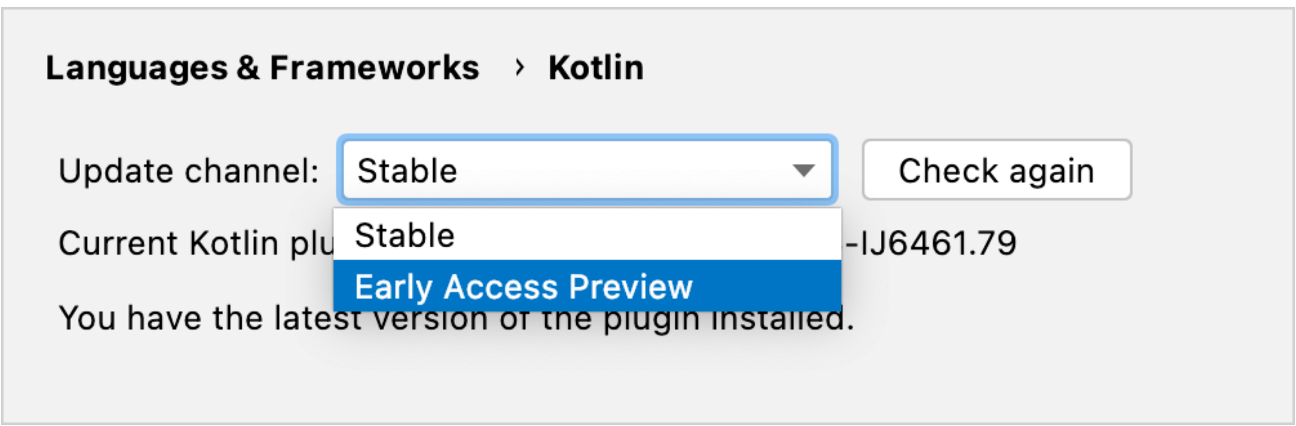
Install the EAP Plugin for IntelliJ IDEA or Android Studio

You can follow these instructions to install the preview version of the Kotlin Plugin for IntelliJ IDEA or Android Studio.

1. Select Tools | Kotlin | Configure Kotlin Plugin Updates.

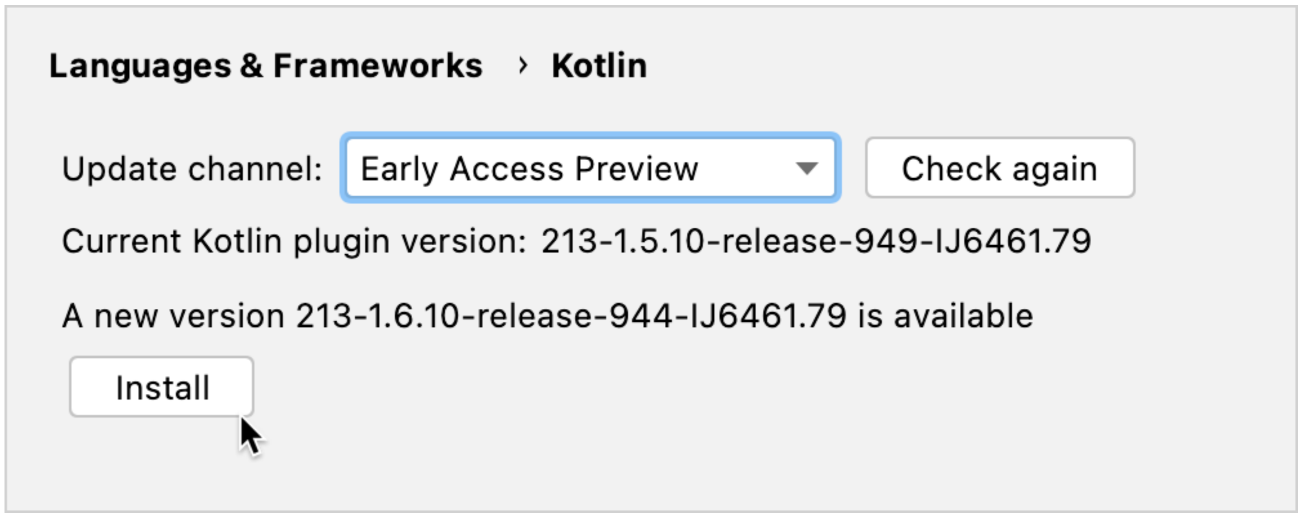
Select Kotlin Plugin Updates

2. In the Update channel list, select the Early Access Preview channel.



Select the EAP update channel

3. Click Check again. The latest EAP build version appears.



Install the EAP build

4. Click Install.

If you want to work on existing projects that were created before installing the EAP version, you need to configure your build for EAP.

If you run into any problems

- Report an issue to our issue tracker, YouTrack.
- Find help in the #eap channel in Kotlin Slack (get an invite).
- Roll back to the latest stable version: in Tools | Kotlin | Configure Kotlin Plugin Updates, select the Stable update channel and click Install.

Configure your build for EAP

If you create new projects using the EAP version of Kotlin, you don't need to perform any additional steps. The Kotlin Plugin will do everything for you!

You only need to configure your build manually for existing projects — projects that were created before installing the EAP version.

To configure your build to use the EAP version of Kotlin, you need to:

- Specify the EAP version of Kotlin. Available EAP versions are listed here.
- Change the versions of dependencies to EAP ones. The EAP version of Kotlin may not work with the libraries of the previously released version.

The following procedures describe how to configure your build in Gradle and Maven:

- Configure in Gradle
- Configure in Maven

Configure in Gradle

This section describes how you can:

- Adjust the Kotlin version
- Adjust versions in dependencies

Adjust the Kotlin version
In the plugins block within build.gradle(.kts), change the KOTLIN-EAP-VERSION to the actual EAP version, such as 1.9.0-RC. Available EAP versions are listed here.

Alternatively, you can specify the EAP version in the pluginManagement block in settings.gradle(.kts) – see Gradle documentation for details.

Here is an example for the Multiplatform project.

Kotlin

```
plugins {
    java
    kotlin("multiplatform") version "KOTLIN-EAP-VERSION"
}

repositories {
    mavenCentral()
}
```

Groovy

```
plugins {
    id 'java'
    id 'org.jetbrains.kotlin.multiplatform' version 'KOTLIN-EAP-VERSION'
}

repositories {
    mavenCentral()
}
```

Adjust versions in dependencies

If you use kotlinx libraries in your project, your versions of the libraries may not be compatible with the EAP version of Kotlin.

To resolve this issue, you need to specify the version of a compatible library in dependencies. For a list of compatible libraries, see EAP build details.

> In most cases we create libraries only for the first EAP version of a specific release and these libraries work with the subsequent EAP versions for this release.
>
> If there are incompatible changes in next EAP versions, we release a new version of the library.

Here is an example.

For the kotlinx.coroutines library, add the version number – 1.7.1 – that is compatible with 1.9.0-RC.

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1")
}
```

Groovy

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1"
}
```

Configure in Maven

In the sample Maven project definition, replace KOTLIN-EAP-VERSION with the actual version, such as 1.9.0-RC. Available EAP versions are listed here.

```
<project ...>
    <properties>
        <kotlin.version>KOTLIN-EAP-VERSION</kotlin.version>
    </properties>

    <repositories>
```

```xml
        <repository>
            <id>mavenCentral</id>
            <url>https://repo1.maven.org/maven2/</url>
        </repository>
    </repositories>

    <pluginRepositories>
        <pluginRepository>
            <id>mavenCentral</id>
            <url>https://repo1.maven.org/maven2/</url>
        </pluginRepository>
    </pluginRepositories>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-stdlib</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>${kotlin.version}</version>
                ...
            </plugin>
        </plugins>
    </build>
</project>
```

FAQ

What is Kotlin?

Kotlin is an open-source statically typed programming language that targets the JVM, Android, JavaScript, Wasm, and Native. It's developed by JetBrains. The project started in 2010 and was open source from very early on. The first official 1.0 release was in February 2016.

What is the current version of Kotlin?

The currently released version is 1.9.0, published on July 6, 2023.
You can find more information on GitHub.

Is Kotlin free?

Yes. Kotlin is free, has been free and will remain free. It is developed under the Apache 2.0 license and the source code is available on GitHub.

Is Kotlin an object-oriented language or a functional one?

Kotlin has both object-oriented and functional constructs. You can use it in both OO and FP styles, or mix elements of the two. With first-class support for features such as higher-order functions, function types and lambdas, Kotlin is a great choice if you're doing or exploring functional programming.

What advantages does Kotlin give me over the Java programming language?

Kotlin is more concise. Rough estimates indicate approximately a 40% cut in the number of lines of code. It's also more type-safe, for example, support for non-nullable types makes applications less prone to NPE's. Other features including smart casting, higher-order functions, extension functions and lambdas with receivers provide the ability to write expressive code as well as facilitating creation of DSL.

Is Kotlin compatible with the Java programming language?

Yes. Kotlin is 100% interoperable with the Java programming language and major emphasis has been placed on making sure that your existing codebase can interact properly with Kotlin. You can easily call Kotlin code from Java and Java code from Kotlin. This makes adoption much easier and lower-risk. There's also an automated Java-to-Kotlin converter built into the IDE that simplifies migration of existing code.

What can I use Kotlin for?

Kotlin can be used for any kind of development, be it server-side, client-side web, Android. With Kotlin/Native currently in the works, support for other platforms such as embedded systems, macOS and iOS is coming. People are using Kotlin for mobile and server-side applications, client-side with JavaScript or JavaFX, and data science, just to name a few possibilities.

Can I use Kotlin for Android development?

Yes. Kotlin is supported as a first-class language on Android. There are hundreds of applications already using Kotlin for Android, such as Basecamp, Pinterest and more. For more information, check out the resource on Android development.

Can I use Kotlin for server-side development?

Yes. Kotlin is 100% compatible with the JVM and as such you can use any existing frameworks such as Spring Boot, vert.x or JSF. In addition, there are specific frameworks written in Kotlin such as Ktor. For more information, check out the resource on server-side development.

Can I use Kotlin for web development?

Yes. In addition to using for backend web, you can also use Kotlin/Wasm for client-side web. Learn how to get started with Kotlin/Wasm.

Can I use Kotlin for desktop development?

Yes. You can use any Java UI framework such as JavaFx, Swing or other. In addition, there are Kotlin specific frameworks such as TornadoFX.

Can I use Kotlin for native development?

Yes. Kotlin/Native is available as a part of Kotlin project. It compiles Kotlin to native code that can run without a VM. You can try it on popular desktop and mobile platforms and even some IoT devices. For more information, check out the Kotlin/Native documentation.

What IDEs support Kotlin?

Kotlin has full out-of-the-box support in IntelliJ IDEA, Android Studio, and JetBrains Fleet with an official Kotlin plugin developed by JetBrains.

Other IDEs and source editors, such as Eclipse, Visual Studio Code, and Atom, have Kotlin community-supported plugins.

You can also try Kotlin Playground for writing, running, and sharing Kotlin code in your browser.

In addition, a command line compiler is available, which provides straightforward support for compiling and running applications.

What build tools support Kotlin?

On the JVM side, the main build tools include Gradle, Maven, Ant, and Kobalt. There are also some build tools available that target client-side JavaScript.

What does Kotlin compile down to?

When targeting the JVM, Kotlin produces Java compatible bytecode.

When targeting JavaScript, Kotlin transpiles to ES5.1 and generates code which is compatible with module systems including AMD and CommonJS.

When targeting native, Kotlin will produce platform-specific code (via LLVM).

Which versions of JVM does Kotlin target?

Kotlin lets you choose the version of JVM for execution. By default, the Kotlin/JVM compiler produces Java 8 compatible bytecode. If you want to make use of optimizations available in newer versions of Java, you can explicitly specify the target Java version from 9 to 20. Note that in this case the resulting bytecode might not run on lower versions. Starting with Kotlin 1.5, the compiler does not support producing bytecode compatible with Java versions below 8.

Is Kotlin hard?

Kotlin is inspired by existing languages such as Java, C#, JavaScript, Scala and Groovy. We've tried to ensure that Kotlin is easy to learn, so that people can easily jump on board, reading and writing Kotlin in a matter of days. Learning idiomatic Kotlin and using some more of its advanced features can take a little longer, but overall it is not a complicated language.
For more information, check out our learning materials.

What companies are using Kotlin?

There are too many companies using Kotlin to list, but some more visible companies that have publicly declared usage of Kotlin, be this via blog posts, GitHub repositories or talks include Square, Pinterest, Basecamp, and Corda.

Who develops Kotlin?

Kotlin is primarily developed by a team of engineers at JetBrains (current team size is 100+). The lead language designer is Roman Elizarov. In addition to the core team, there are also over 250 external contributors on GitHub.

Where can I learn more about Kotlin?

The best place to start is our website. From there you can download the compiler, try it online as well as get access to resources.

Are there any books on Kotlin?

There are a number of books available for Kotlin. Some of them we have reviewed and can recommend to start with. They are listed on the Books page. For more books, see the community-maintained list at kotlin.link.

Are any online courses available for Kotlin?

You can learn all the Kotlin essentials while creating working applications with the Kotlin Core track by JetBrains Academy.

A few other courses you can take:

- Pluralsight Course: Getting Started with Kotlin by Kevin Jones
- O'Reilly Course: Introduction to Kotlin Programming by Hadi Hariri
- Udemy Course: 10 Kotlin Tutorials for Beginneres by Peter Sommerhoff

You can also check out the other tutorials and content on our YouTube channel.

Does Kotlin have a community?

Yes! Kotlin has a very vibrant community. Kotlin developers hang out on the Kotlin forums, StackOverflow and more actively on the Kotlin Slack (with close to 30000 members as of April 2020).

Are there Kotlin events?

Yes! There are many User Groups and Meetups now focused exclusively around Kotlin. You can find a list on the website. In addition, there are community-organized Kotlin Nights events around the world.

Is there a Kotlin conference?

Yes! KotlinConf is an annual conference hosted by JetBrains, which brings together developers, enthusiasts, and experts from around the world to share their knowledge and experience with Kotlin.

In addition to technical talks and workshops, KotlinConf also offers networking opportunities, community interactions, and social events where attendees can connect with fellow Kotliners and exchange ideas. It serves as a platform for fostering collaboration and community building within the Kotlin ecosystem.

Kotlin is also being covered in different conferences worldwide. You can find a list of upcoming talks on the website.

Is Kotlin on social media?

Yes. Subscribe to the Kotlin YouTube channel and follow Kotlin on Twitter.

Any other online Kotlin resources?

The website has a bunch of online resources, including Kotlin Digests by community members, a newsletter, a podcast and more.

Where can I get an HD Kotlin logo?

Logos can be downloaded here. When using the logos, please follow simple rules in the guidelines.pdf inside the archive and Kotlin brand usage guidelines.

For more information, check out the page about Kotlin brand assets.

Kotlin Evolution

Principles of Pragmatic Evolution

> Language design is cast in stone,
>
> but this stone is reasonably soft,
>
> and with some effort we can reshape it later.
>
> Kotlin Design Team

Kotlin is designed to be a pragmatic tool for programmers. When it comes to language evolution, its pragmatic nature is captured by the following principles:

- Keep the language modern over the years.
- Stay in the constant feedback loop with the users.
- Make updating to new versions comfortable for the users.

As this is key to understanding how Kotlin is moving forward, let's expand on these principles.

Keeping the Language Modern. We acknowledge that systems accumulate legacy over time. What had once been cutting-edge technology can be hopelessly outdated today. We have to evolve the language to keep it relevant to the needs of the users and up-to-date with their expectations. This includes not only adding new features, but also phasing out old ones that are no longer recommended for production use and have altogether become legacy.

Comfortable Updates. Incompatible changes, such as removing things from a language, may lead to painful migration from one version to the next if carried out without proper care. We will always announce such changes well in advance, mark things as deprecated and provide automated migration tools before the change happens. By the time the language is changed we want most of the code in the world to be already updated and thus have no issues migrating to the new version.

Feedback Loop. Going through deprecation cycles requires significant effort, so we want to minimize the number of incompatible changes we'll be making in the future. Apart from using our best judgement, we believe that trying things out in real life is the best way to validate a design. Before casting things in stone we want them battle-tested. This is why we use every opportunity to make early versions of our designs available in production versions of the language, but in one of the pre-stable statuses: Experimental, Alpha, or Beta. Such features are not stable, they can be changed at any time, and the users that opt into using them do so

explicitly to indicate that they are ready to deal with the future migration issues. These users provide invaluable feedback that we gather to iterate on the design and make it rock-solid.

Incompatible changes

If, upon updating from one version to another, some code that used to work doesn't work any more, it is an incompatible change in the language (sometimes referred to as "breaking change"). There can be debates as to what "doesn't work any more" means precisely in some cases, but it definitely includes the following:

- Code that compiled and ran fine is now rejected with an error (at compile or link time). This includes removing language constructs and adding new restrictions.
- Code that executed normally is now throwing an exception.

The less obvious cases that belong to the "grey area" include handling corner cases differently, throwing an exception of a different type than before, changing behavior observable only through reflection, changes in undocumented/undefined behavior, renaming binary artifacts, etc. Sometimes such changes are very important and affect migration experience dramatically, sometimes they are insignificant.

Some examples of what definitely isn't an incompatible change include

- Adding new warnings.
- Enabling new language constructs or relaxing limitations for existing ones.
- Changing private/internal APIs and other implementation details.

The principles of Keeping the Language Modern and Comfortable Updates suggest that incompatible changes are sometimes necessary, but they should be introduced carefully. Our goal is to make the users aware of upcoming changes well in advance to let them migrate their code comfortably.

Ideally, every incompatible change should be announced through a compile-time warning reported in the problematic code (usually referred to as a deprecation warning) and accompanied with automated migration aids. So, the ideal migration workflow goes as follows:

- Update to version A (where the change is announced)
  - See warnings about the upcoming change
  - Migrate the code with the help of the tooling
- Update to version B (where the change happens)
  - See no issues at all

In practice some changes can't be accurately detected at compile time, so no warnings can be reported, but at least the users will be notified through Release notes of version A that a change is coming in version B.

Dealing with compiler bugs

Compilers are complicated software and despite the best effort of their developers they have bugs. The bugs that cause the compiler itself to fail or report spurious errors or generate obviously failing code, though annoying and often embarrassing, are easy to fix, because the fixes do not constitute incompatible changes. Other bugs may cause the compiler to generate incorrect code that does not fail: e.g. by missing some errors in the source or simply generating wrong instructions. Fixes of such bugs are technically incompatible changes (some code used to compile fine, but now it won't any more), but we are inclined to fixing them as soon as possible to prevent the bad code patterns from spreading across user code. In our opinion, this serves the principle of Comfortable Updates, because fewer users have a chance of encountering the issue. Of course, this applies only to bugs that are found soon after appearing in a released version.

Decision making

JetBrains, the original creator of Kotlin, is driving its progress with the help of the community and in accord with the Kotlin Foundation.

All changes to the Kotlin Programming Language are overseen by the Lead Language Designer (currently Roman Elizarov). The Lead Designer has the final say in all matters related to language evolution. Additionally, incompatible changes to fully stable components have to be approved to by the Language Committee designated under the Kotlin Foundation (currently comprised of Jeffrey van Gogh, William R. Cook and Roman Elizarov).

The Language Committee makes final decisions on what incompatible changes will be made and what exact measures should be taken to make user updates comfortable. In doing so, it relies on a set of guidelines available here.

Feature releases and incremental releases

Stable releases with versions 1.2, 1.3, etc. are usually considered to be feature releases bringing major changes in the language. Normally, we publish incremental releases, numbered 1.2.20, 1.2.30, etc, in between feature releases.

Incremental releases bring updates in the tooling (often including features), performance improvements and bug fixes. We try to keep such versions compatible with each other, so changes to the compiler are mostly optimizations and warning additions/removals. Pre-stable features may, of course, be added, removed or changed at any time.

Feature releases often add new features and may remove or change previously deprecated ones. Feature graduation from pre-stable to stable also happens in feature releases.

EAP builds

Before releasing stable versions, we usually publish a number of preview builds dubbed EAP (for "Early Access Preview") that let us iterate faster and gather feedback from the community. EAPs of feature releases usually produce binaries that will be later rejected by the stable compiler to make sure that possible bugs in the binary format survive no longer than the preview period. Final Release Candidates normally do not bear this limitation.

Pre-stable features

According to the Feedback Loop principle described above, we iterate on our designs in the open and release versions of the language where some features have one of the pre-stable statuses and are supposed to change. Such features can be added, changed or removed at any point and without warning. We do our best to ensure that pre-stable features can't be used accidentally by an unsuspecting user. Such features usually require some sort of an explicit opt-in either in the code or in the project configuration.

Pre-stable features usually graduate to the stable status after some iterations.

Status of different components

To check the stability status of different components of Kotlin (Kotlin/JVM, JS, Native, various libraries, etc), please consult this link.

Libraries

A language is nothing without its ecosystem, so we pay extra attention to enabling smooth library evolution.

Ideally, a new version of a library can be used as a "drop-in replacement" for an older version. This means that upgrading a binary dependency should not break anything, even if the application is not recompiled (this is possible under dynamic linking).

On the one hand, to achieve this, the compiler has to provide certain ABI stability guarantees under the constraints of separate compilation. This is why every change in the language is examined from the point of view of binary compatibility.

On the other hand, a lot depends on the library authors being careful about which changes are safe to make. Thus it's very important that library authors understand how source changes affect compatibility and follow certain best practices to keep both APIs and ABIs of their libraries stable. Here are some assumptions that we make when considering language changes from the library evolution standpoint:

- Library code should always specify return types of public/protected functions and properties explicitly thus never relying on type inference for public API. Subtle changes in type inference may cause return types to change inadvertently, leading to binary compatibility issues.
- Overloaded functions and properties provided by the same library should do essentially the same thing. Changes in type inference may result in more precise static types to be known at call sites causing changes in overload resolution.

Library authors can use the @Deprecated and @RequiresOptIn annotations to control the evolution of their API surface. Note that @Deprecated(level=HIDDEN) can be used to preserve binary compatibility even for declarations removed from the API.

Also, by convention, packages named "internal" are not considered public API. All API residing in packages named "experimental" is considered pre-stable and can change at any moment.

We evolve the Kotlin Standard Library (kotlin-stdlib) for stable platforms according to the principles stated above. Changes to the contracts for its API undergo the same procedures as changes in the language itself.

Compiler keys

Command line keys accepted by the compiler are also a kind of public API, and they are subject to the same considerations. Supported flags (those that don't have the "-X" or "-XX" prefix) can be added only in feature releases and should be properly deprecated before removing them. The "-X" and "-XX" flags are experimental and can be added and removed at any time.

Compatibility tools

As legacy features get removed and bugs fixed, the source language changes, and old code that has not been properly migrated may not compile any more. The normal deprecation cycle allows a comfortable period of time for migration, and even when it's over and the change ships in a stable version, there's still a way to compile unmigrated code.

Compatibility flags

We provide the -language-version X.Y and -api-version X.Y flags that make a new version emulate the behavior of an old one for compatibility purposes. To give you more time for migration, we support the development for three previous language and API versions in addition to the latest stable one.

Actively maintained code bases can benefit from getting bug fixes ASAP, without waiting for a full deprecation cycle to complete. Currently, such project can enable the -progressive flag and get such fixes enabled even in incremental releases.

All flags are available on the command line as well as Gradle and Maven.

Evolving the binary format

Unlike sources that can be fixed by hand in the worst case, binaries are a lot harder to migrate, and this makes backwards compatibility very important in the case of binaries. Incompatible changes to binaries can make updates very uncomfortable and thus should be introduced with even more care than those in the source language syntax.

For fully stable versions of the compiler the default binary compatibility protocol is the following:

- All binaries are backwards compatible, i.e. a newer compiler can read older binaries (e.g. 1.3 understands 1.0 through 1.2),
- Older compilers reject binaries that rely on new features (e.g. a 1.0 compiler rejects binaries that use coroutines).
- Preferably (but we can't guarantee it), the binary format is mostly forwards compatible with the next feature release, but not later ones (in the cases when new features are not used, e.g. 1.3 can understand most binaries from 1.4, but not 1.5).

This protocol is designed for comfortable updates as no project can be blocked from updating its dependencies even if it's using a slightly outdated compiler.

Please note that not all target platforms have reached this level of stability (but Kotlin/JVM has).

Stability of Kotlin components

The Kotlin language and toolset are divided into many components such as the compilers for the JVM, JS and Native targets, the Standard Library, various accompanying tools and so on. Many of these components were officially released as Stable which means that they are evolved in the backward-compatible way following the principles of Comfortable Updates and Keeping the Language Modern. Among such stable components are, for example, the Kotlin compiler for the JVM, the Standard Library, and Coroutines.

Following the Feedback Loop principle we release many things early for the community to try out, so a number of components are not yet released as Stable. Some of them are very early stage, some are more mature. We mark them as Experimental, Alpha or Beta depending on how quickly each component is evolving and how much risk the users are taking when adopting it.

Stability levels explained

Here's a quick guide to these stability levels and their meaning:

Experimental means "try it only in toy projects":

- We are just trying out an idea and want some users to play with it and give feedback. If it doesn't work out, we may drop it any minute.

Alpha means "use at your own risk, expect migration issues":

- We decided to productize this idea, but it hasn't reached the final shape yet.

Beta means "you can use it, we'll do our best to minimize migration issues for you":

- It's almost done, user feedback is especially important now.
- Still, it's not 100% finished, so changes are possible (including ones based on your own feedback).

* Watch for deprecation warnings in advance for the best update experience.

We collectively refer to Experimental, Alpha and Beta as pre-stable levels.

Stable means "use it even in most conservative scenarios":

* It's done. We will be evolving it according to our strict backward compatibility rules.

Please note that stability levels do not say anything about how soon a component will be released as Stable. Similarly, they do not indicate how much a component will be changed before release. They only say how fast a component is changing and how much risk of update issues users are running.

GitHub badges for Kotlin components

The Kotlin GitHub organization hosts different Kotlin-related projects. Some of them we develop full-time, while others are side projects.

Each Kotlin project has two GitHub badges describing its stability and support status:

* Stability status. This shows how quickly each project is evolving and how much risk the users are taking when adopting it. The stability status completely coincides with the stability level of the Kotlin language features and its components:
  * experimental stands for Experimental
  * alpha stands for Alpha
  * beta stands for Beta
  * stable stands for Stable
* Support status. This shows our commitment to maintaining a project and helping users to solve their problems. The level of support is unified for all JetBrains products.
  See the JetBrains Confluence document for details.

Stability of subcomponents

A stable component may have an experimental subcomponent, for example:

* a stable compiler may have an experimental feature;
* a stable API may include experimental classes or functions;
* a stable command-line tool may have experimental options.

We make sure to document precisely which subcomponents are not stable. We also do our best to warn users where possible and ask to opt in explicitly to avoid accidental usages of features that have not been released as stable.

Current stability of Kotlin components

Component	Status	Status since version	Comment
Kotlin/JVM	Stable	1.0	
Kotlin K2 (JVM)	Alpha	1.7	
kotlin-stdlib (JVM)	Stable	1.0	
Coroutines	Stable	1.3	

Component	Status	Status since version	Comment
kotlin-reflect (JVM)	Beta	1.0	
Kotlin/JS (Classic back-end)	Stable	1.3	Deprecated from 1.8.0, read the IR migration guide
Kotlin/JVM (IR-based)	Stable	1.5	
Kotlin/JS (IR-based)	Stable	1.8	
Kotlin/Native Runtime	Beta	1.3	
Kotlin/Native memory manager	Beta	1.7.20	Same stability level as Kotlin/Native
klib binaries	Beta	1.9.0	
Kotlin Multiplatform	Beta	1.7.20	
Kotlin/Native interop with C and Objective C	Beta	1.3	
CocoaPods integration	Beta	1.3	
Kotlin Multiplatform Mobile plugin for Android Studio	Beta	0.5.2	Versioned separately from the language
expect/actual language feature	Beta	1.2	
KDoc syntax	Stable	1.0	
Dokka	Beta	1.6	
Scripting syntax and semantics	Alpha	1.2	
Scripting embedding and extension API	Beta	1.5	
Scripting IDE support	Beta		Available since IntelliJ IDEA 2023.1 and later
CLI scripting	Alpha	1.2	
Compiler Plugin API	Experimental	1.0	

Component	Status	Status since version	Comment
Serialization Compiler Plugin	Stable	1.4	
Serialization Core Library	Stable	1.0.0	Versioned separately from the language
Inline classes	Stable	1.5	
Unsigned arithmetic	Stable	1.5	
Contracts in stdlib	Stable	1.3	
User-defined contracts	Experimental	1.3	
All other experimental components, by default	Experimental	N/A	

The pre-1.4 version of this page is available here.

Stability of Kotlin components (pre 1.4)

There can be different modes of stability depending of how quickly a component is evolving:

- Moving fast (MF): no compatibility should be expected between even incremental releases, any functionality can be added, removed or changed without warning.
- Additions in Incremental Releases (AIR): things can be added in an incremental release, removals and changes of behavior should be avoided and announced in a previous incremental release if necessary.
- Stable Incremental Releases (SIR): incremental releases are fully compatible, only optimizations and bug fixes happen. Any changes can be made in a feature release.
- Fully Stable (FS): incremental releases are fully compatible, only optimizations and bug fixes happen. Feature releases are backwards compatible.

Source and binary compatibility may have different modes for the same component, e.g. the source language can reach full stability before the binary format stabilizes, or vice versa.

The provisions of the Kotlin evolution policy fully apply only to components that have reached Full Stability (FS). From that point on incompatible changes have to be approved by the Language Committee.

Component	Status Entered at version	Mode for Sources	Mode for Binaries
Kotlin/JVM	1.0	FS	FS
kotlin-stdlib (JVM)	1.0	FS	FS
KDoc syntax	1.0	FS	N/A
Coroutines	1.3	FS	FS

Component	Status Entered at version	Mode for Sources	Mode for Binaries
kotlin-reflect (JVM)	1.0	SIR	SIR
Kotlin/JS	1.1	AIR	MF
Kotlin/Native	1.3	AIR	MF
Kotlin Scripts (*.kts)	1.2	AIR	MF
dokka	0.1	MF	N/A
Kotlin Scripting APIs	1.2	MF	MF
Compiler Plugin API	1.0	MF	MF
Serialization	1.3	MF	MF
Multiplatform Projects	1.2	MF	MF
Inline classes	1.3	MF	MF
Unsigned arithmetics	1.3	MF	MF
All other experimental features, by default	N/A	MF	MF

Compatibility guide for Kotlin 1.9

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.8 to Kotlin 1.9.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Prohibit super constructor call when the super interface type is a function literal

Issue: KT-46344

Component: Core language

Incompatible change type: source

Short summary: If an interface inherits from a function literal type, Kotlin 1.9 prohibits super constructor calls because no such constructor exists.

Deprecation cycle:

- 1.7.0: report a warning (or an error in progressive mode)
- 1.9.0: raise the warning to an error

Prohibit cycles in annotation parameter types

Issue: KT-47932

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 prohibits the type of an annotation being used as one of its parameter types, either directly or indirectly. This prevents cycles from being created. However, you are allowed to have parameter types that are an Array or a vararg of the annotation type.

Deprecation cycle:

- 1.7.0: report a warning (or an error in progressive mode) on cycles in types of annotation parameters
- 1.9.0: raise the warning to an error, -XXLanguage:-ProhibitCyclesInAnnotations can be used to temporarily revert to pre-1.9 behavior

Prohibit use of @ExtensionFunctionType annotation on function types with no parameters

Issue: KT-43527

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 prohibits using the @ExtensionFunctionType annotation on function types with no parameters, or on types that aren't function types.

Deprecation cycle:

- 1.7.0: report a warning for annotations on types that aren't function types, report an error for annotations on types that are function types
- 1.9.0: raise the warning for function types to an error

Prohibit Java field type mismatch on assignment

Issue: KT-48994

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.9 reports a compiler error if it detects that the type of a value assigned to a Java field doesn't match the Java field's projected type.

Deprecation cycle:

- 1.6.0: report a warning (or an error in the progressive mode) when a projected Java field type doesn't match the assigned value type
- 1.9.0: raise the warning to an error, -XXLanguage:-RefineTypeCheckingOnAssignmentsToJavaFields can be used to temporarily revert to pre-1.9 behavior

No source code excerpts in platform-type nullability assertion exceptions

Issue: KT-57570

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: In Kotlin 1.9, exception messages for expression null checks do not include source code excerpts. Instead, the name of the method or field is displayed. If the expression is not a method or field, there is no additional information provided in the message.

Deprecation cycle:

- < 1.9.0: exception messages generated by expression null checks contain source code excerpts
- 1.9.0: exception messages generated by expression null checks contain method or field names only, -XXLanguage:-NoSourceCodeInNotNullAssertionExceptions can be used to temporarily revert to pre-1.9 behavior

Prohibit the delegation of super calls to an abstract superclass member

Issues: KT-45508, KT-49017, KT-38078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will report a compile error when an explicit or implicit super call is delegated to an abstract member of the superclass, even if there's a default implementation in a super interface.

Deprecation cycle:

- 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
- 1.7.0: report a warning if a super call, in fact, accesses an abstract member from a superclass
- 1.7.0: report an error in all affected cases if the -Xjvm-default=all or -Xjvm-default=all-compatibility compatibility modes are enabled; report an error in the progressive mode
- 1.8.0: report an error in cases of declaring a concrete class with a non-overridden abstract method from the superclass, and super calls of Any methods are overridden as abstract in the superclass
- 1.9.0: report an error in all affected cases, including explicit super calls to an abstract method from the super class

Deprecate confusing grammar in when-with-subject

Issue: KT-48385

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 deprecated several confusing grammar constructs in when condition expressions.

Deprecation cycle:

- 1.6.20: introduce a deprecation warning on the affected expressions
- 1.8.0: raise this warning to an error, -XXLanguage:-ProhibitConfusingSyntaxInWhenBranches can be used to temporarily revert to the pre-1.8 behavior
- >= 2.1: repurpose some deprecated constructs for new language features

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type is needed semantically.

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, -Xuse-old-backend can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 2.0: fix the downcast behavior in other affected cases

Prohibit upper bound violation in a generic type alias usage (a type parameter used in a generic type argument of a type argument of the aliased type)

Issue: KT-54066

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when the typealias type parameter is used as a generic type argument of a type argument of the aliased type, for example, typealias Alias<T> = Base<List<T>>.

Deprecation cycle:

- 1.8.0: report a warning when a generic typealias usage has type arguments violating upper bound constraints of the corresponding type parameters of the aliased type
- 2.0.0: raise the warning to an error

Keep nullability when approximating local types in public signatures

Issue: KT-53982

Component: Core language

Incompatible change type: source, binary

Short summary: when a local or anonymous type is returned from an expression-body function without an explicitly specified return type, Kotlin compiler infers (or approximates) the return type using the known supertype of that type. During this, the compiler can infer a non-nullable type where the null value could in fact be returned.

Deprecation cycle:

- 1.8.0: approximate flexible types by flexible supertypes
- 1.8.0: report a warning when a declaration is inferred to have a non-nullable type that should be nullable, prompting users to specify the type explicitly
- 2.0.0: approximate nullable types by nullable supertypes, -XXLanguage:-KeepNullabilityWhenApproximatingLocalType can be used to temporarily revert to the pre-2.0 behavior

Do not propagate deprecation through overrides

Issue: KT-47902

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will no longer propagate deprecation from a deprecated member in the superclass to its overriding member in the subclass, thus providing an explicit mechanism for deprecating a member of the superclass while leaving it non-deprecated in the subclass.

Deprecation cycle:

- 1.6.20: reporting a warning with the message of the future behavior change and a prompt to either suppress this warning or explicitly write a @Deprecated annotation on an override of a deprecated member
- 1.9.0: stop propagating deprecation status to the overridden members. This change also takes effect immediately in the progressive mode

Prohibit using collection literals in annotation classes anywhere except their parameters declaration

Issue: KT-39041

Component: Core language

Incompatible change type: source

Short summary: Kotlin allows using collection literals in a restricted way - for passing arrays to parameters of annotation classes or specifying default values for these parameters. However besides that, Kotlin allowed using collections literals anywhere else inside an annotation class, for example, in its nested object. Kotlin 1.9 will prohibit using collection literals in annotation classes anywhere except their parameters' default values.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on array literals in nested objects in annotation classes
- 1.9.0: raise the warning to an error

Prohibit forward referencing of parameters in default value expressions

Issue: KT-25694

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit forward referencing of parameters in default value expressions of other parameters. This ensures that by the time the parameter is accessed in a default value expression, it would already have a value either passed to the function or initialized by its own default value expression.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when a parameter with default value is references in default value of another parameter that comes before it
- 1.9.0: raise the warning to an error, -XXLanguage:-ProhibitIllegalValueParameterUsageInDefaultArguments can be used to temporarily revert to the pre-1.9 behavior

Prohibit extension calls on inline functional parameters

Issue: KT-52502

Component: Core language

Incompatible change type: source

Short summary: while Kotlin allowed passing an inline functional parameter to another inline function as a receiver, it always resulted in compiler exceptions when compiling such code. Kotlin 1.9 will prohibit this, thus reporting an error instead of crashing the compiler.

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) for inline extension calls on inline functional parameters
- 1.9.0: raise the warning to an error

Prohibit calls to infix functions named suspend with an anonymous function argument

Issue: KT-49264

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will no longer allow calling infix functions named suspend that have a single argument of a functional type passed as an anonymous function literal.

Deprecation cycle:

- 1.7.20: report a warning on suspend infix calls with an anonymous function literal
- 1.9.0: raise the warning to an error, -XXLanguage:-ModifierNonBuiltinSuspendFunError can be used to temporarily revert to the pre-1.9 behavior
- TODO: Change how the suspend fun token sequence is interpreted by the parser

Prohibit using captured type parameters in inner classes against their variance

Issue: KT-50947

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit using type parameters of an outer class having in or out variance in an inner class of that class in positions violating that type parameters' declared variance.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when an outer class' type parameter usage position violates the variance rules of that parameter
- 1.9.0: raise the warning to an error, -XXLanguage:-ReportTypeVarianceConflictOnQualifierArguments can be used to temporarily revert to the pre-1.9 behavior

Prohibit recursive call of a function without explicit return type in compound assignment operators

Issue: KT-48546

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit calling a function without explicitly specified return type in an argument of a compound assignment operator inside that function's body, as it currently does in other expressions inside the body of that function.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when a function without explicitly specified return type is called recursively in that function's body in a compound assignment operator argument
- 1.9.0: raise the warning to an error

Prohibit unsound calls with expected @NotNull T and given Kotlin generic parameter with nullable bound

Issue: KT-36770

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit method calls where a value of a potentially nullable generic type is passed for a @NotNull-annotated parameter of a Java method.

Deprecation cycle:

- 1.5.20: report a warning when an unconstrained generic type parameter is passed where a non-nullable type is expected
- 1.9.0: report a type mismatch error instead of the warning above, -XXLanguage:-ProhibitUsingNullableTypeParameterAgainstNotNullAnnotated can be used to temporarily revert to the pre-1.8 behavior

Prohibit access to members of a companion of an enum class from entry initializers of this enum

Issue: KT-49110

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit all kinds of access to the companion object of an enum from an enum entry initializer.

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) on such companion member access
- 1.9.0: raise the warning to an error, -XXLanguage:-ProhibitAccessToEnumCompanionMembersInEnumConstructorCall can be used to temporarily revert to the pre-1.8 behavior

Deprecate and remove Enum.declaringClass synthetic property

Issue: KT-49653

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin allowed using the synthetic property declaringClass on Enum values produced from the method getDeclaringClass() of the underlying Java class java.lang.Enum even though this method is not available for Kotlin Enum type. Kotlin 1.9 will prohibit using this property, proposing to migrate to the extension property declaringJavaClass instead.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on declaringClass property usages, propose the migration to declaringJavaClass extension
- 1.9.0: raise the warning to an error, -XXLanguage:-ProhibitEnumDeclaringClass can be used to temporarily revert to the pre-1.9 behavior
- 2.0.0: remove declaringClass synthetic property

Deprecate enable and compatibility modes of the compiler option -Xjvm-default

Issues: KT-46329, KT-54746

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.9 prohibits using the enable and compatibility modes of the -Xjvm-default compiler option.

Deprecation cycle:

- 1.6.20: introduce a warning on the enable and compatibility modes of the -Xjvm-default compiler option
- 1.9.0: raise this warning to an error

Prohibit implicit inferring a type variable into an upper bound in the builder inference context

Issue: KT-47986

Component: Core language

Incompatible change type: source

Short summary: Kotlin 2.0 will prohibit inferring a type variable into the corresponding type parameter's upper bound in the absence of any use-site type information in the scope of builder inference lambda functions, the same way as it does currently in other contexts.

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) when a type parameter is inferred into declared upper bounds in the absence of use-site type information
- 2.0.0: raise the warning to an error

Standard library

Warn about potential overload resolution change when Range/Progression starts implementing Collection

Issue: KT-49276

Component: Core language / kotlin-stdlib

Incompatible change type: source

Short summary: it is planned to implement the Collection interface in the standard progressions and concrete ranges inherited from them in Kotlin 1.9. This could make a different overload selected in the overload resolution if there are two overloads of some method, one accepting an element and another accepting a collection. Kotlin will make this situation visible by reporting a warning or an error when such overloaded method is called with a range or progression argument.

Deprecation cycle:

- 1.6.20: report a warning when an overloaded method is called with the standard progression or its range inheritor as an argument if implementing the Collection interface by this progression/range leads to another overload being selected in this call in future
- 1.8.0: raise this warning to an error
- 2.1.0: stop reporting the error, implement Collection interface in progressions thus changing the overload resolution result in the affected cases

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the kotlin.dom and kotlin.browser packages are moved to the corresponding kotlinx.* packages to prepare for extracting them from stdlib.

Deprecation cycle:

- 1.4.0: introduce the replacement API in kotlinx.dom and kotlinx.browser packages
- 1.4.0: deprecate the API in kotlin.dom and kotlin.browser packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- 1.8.20: remove the deprecated functions from stdlib for JS-IR target
- >= 2.0: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: String.concat(String), String.match(regex: String), String.matches(regex: String), and the sort functions on arrays taking a comparison function, for example, Array<out T>.sort(comparison: (a: T, b: T) -> Int).

Deprecation cycle:

- 1.6.0: deprecate the affected functions with a warning
- 1.9.0: raise the deprecation level to an error
- >=2.0: remove the deprecated functions from the public API

Tools

Remove enableEndorsedLibs flag from Gradle setup

Issue: KT-54098

Component: Gradle

Incompatible change type: source

Short summary: the enableEndorsedLibs flag is no longer supported in Gradle setup.

Deprecation cycle:

- < 1.9.0: enableEndorsedLibs flag is supported in Gradle setup
- 1.9.0: enableEndorsedLibs flag is not supported in Gradle setup

Remove Gradle conventions

> Issue: KT-52976
>
> Component: Gradle
>
> Incompatible change type: source
>
> Short summary: Gradle conventions were deprecated in Gradle 7.1 and have been removed in Gradle 8.
>
> Deprecation cycle:
>
> - 1.7.20: Gradle conventions deprecated
> - 1.9.0: Gradle conventions removed

Remove classpath property of KotlinCompile task

> Issue: KT-53748
>
> Component: Gradle
>
> Incompatible change type: source
>
> Short summary: the classpath property of the KotlinCompile task is removed.
>
> Deprecation cycle:
>
> - 1.7.0: the classpath property is deprecated
> - 1.8.0: raise the deprecation level to an error
> - 1.9.0: remove the deprecated functions from the public API

Deprecate kotlin.internal.single.build.metrics.file property

> Issue: KT-53357
>
> Component: Gradle
>
> Incompatible change type: source
>
> Short summary: deprecate the kotlin.internal.single.build.metrics.file property used to define a single file for build reports. Use the property kotlin.build.report.single_file instead with kotlin.build.report.output=single_file.
>
> Deprecation cycle:
>
> - 1.8.0: raise the deprecation level to a warning
> - >= 1.9: delete the property

Compatibility guide for Kotlin 1.8

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.7 to Kotlin 1.8.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Prohibit the delegation of super calls to an abstract superclass member

> Issues: KT-45508, KT-49017, KT-38078
>
> Component: Core language
>
> Incompatible change type: source
>
> Short summary: Kotlin will report a compile error when an explicit or implicit super call is delegated to an abstract member of the superclass, even if there's a default implementation in a super interface
>
> Deprecation cycle:
>
> - 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
> - 1.7.0: report a warning if a super call, in fact, accesses an abstract member from a superclass
> - 1.7.0: report an error in all affected cases if the -Xjvm-default=all or -Xjvm-default=all-compatibility compatibility modes are enabled; report an error in the progressive mode
> - 1.8.0: report an error in cases of declaring a concrete class with a non-overridden abstract method from the superclass, and super calls of Any methods are overridden as abstract in the superclass
> - 1.9.0: report an error in all affected cases, including explicit super calls to an abstract method from the super class

Deprecate confusing grammar in when-with-subject

> Issue: KT-48385
>
> Component: Core language
>
> Incompatible change type: source
>
> Short summary: Kotlin 1.6 deprecated several confusing grammar constructs in when condition expressions
>
> Deprecation cycle:
>
> - 1.6.20: introduce a deprecation warning on the affected expressions
> - 1.8.0: raise this warning to an error, -XXLanguage:-ProhibitConfusingSyntaxInWhenBranches can be used to temporarily revert to the pre-1.8 behavior
> - >= 1.9: repurpose some deprecated constructs for new language features

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type is needed semantically

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, -Xuse-old-backend can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 1.9: fix the downcast behavior in other affected cases

Make private constructors of sealed classes really private

Issue: KT-44866

Component: Core language

Incompatible change type: source

Short summary: after relaxing restrictions on where the inheritors of sealed classes could be declared in the project structure, the default visibility of sealed class constructors became protected. However, until 1.8, Kotlin still allowed calling explicitly declared private constructors of sealed classes outside those classes' scopes

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) when a private constructor of a sealed class is called outside that class
- 1.8.0: use default visibility rules for private constructors (a call to a private constructor can be resolved only if this call is inside the corresponding class), the old behavior can be brought back temporarily by specifying the -XXLanguage:-UseConsistentRulesForPrivateConstructorsOfSealedClasses compiler argument

Prohibit using operator == on incompatible numeric types in builder inference context

Issue: KT-45508

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit using the operator == on incompatible numeric types, for example, Int and Long, in scopes of builder inference lambda functions, the same way as it currently does in other contexts

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) when the operator == is used on incompatible numeric types
- 1.8.0: raise the warning to an error, -XXLanguage:-ProperEqualityChecksInBuilderInferenceCalls can be used to temporarily revert to the pre-1.8 behavior

Prohibit if without else and non-exhaustive when in right hand side of elvis operator

Issue: KT-44705

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit using a non-exhaustive when or the if expression without an else branch on the right hand side of the Elvis operator (?:). Previously, it was allowed if the Elvis operator's result was not used as an expression

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) on such non-exhaustive if and when expressions
- 1.8.0: raise this warning to an error,
  -XXLanguage:-ProhibitNonExhaustiveIfInRhsOfElvis can be used to temporarily revert to the pre-1.8 behavior

Prohibit upper bound violation in a generic type alias usage (one type parameter used in several type arguments of the aliased type)

Issues: KT-29168

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when one typealias type parameter is used in several type arguments of the aliased type, for example, typealias Alias<T> = Base<T, T>

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on usages of a type alias with type arguments violating upper bound constraints of the corresponding type parameters of the aliased type
- 1.8.0: raise this warning to an error, -XXLanguage:-ReportMissingUpperBoundsViolatedErrorOnAbbreviationAtSupertypes can be used to temporarily revert to the pre-1.8 behavior

Prohibit upper bound violation in a generic type alias usage (a type parameter used in a generic type argument of a type argument of the aliased type)

Issue: KT-54066

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when the typealias type parameter is used as a generic type argument of a type argument of the aliased type, for example, typealias Alias<T> = Base<List<T>>

Deprecation cycle:

- 1.8.0: report a warning when a generic typealias usage has type arguments violating upper bound constraints of the corresponding type parameters of the aliased type
- >=1.10: raise the warning to an error

Prohibit using a type parameter declared for an extension property inside delegate

Issue: KT-24643

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit delegating extension properties on a generic type to generic types that use the type parameter of the receiver in an unsafe way

Deprecation cycle:

- 1.6.0: report a warning (or an error in the progressive mode) when delegating an extension property to a type that uses type parameters inferred from the delegated property's type arguments in a particular way
- 1.8.0: raise the warning to an error, -XXLanguage:-ForbidUsingExtensionPropertyTypeParameterInDelegate can be used to temporarily revert to the pre-1.8 behavior

Forbid @Synchronized annotation on suspend functions

Issue: KT-48516

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit placing the @Synchronized annotation on suspend functions because a suspending call should not be allowed to happen inside a synchronized block

Deprecation cycle:

- 1.6.0: report a warning on suspend functions annotated with the @Synchronized annotation, the warning is reported as an error in the progressive mode
- 1.8.0: raise the warning to an error, -XXLanguage:-SynchronizedSuspendError can be used to temporarily revert to the pre-1.8 behavior

Prohibit using spread operator for passing arguments to non-vararg parameters

Issue: KT-48162

Component: Core language

Incompatible change type: source

Short summary: Kotlin allowed passing arrays with the spread operator (*) to non-vararg array parameters in certain conditions. Since Kotlin 1.8, this will be prohibited

Deprecation cycle:

- 1.6.0: report a warning (or an error in the progressive mode) on using the spread operator where a non-vararg array parameter is expected
- 1.8.0: raise the warning to an error, -XXLanguage:-ReportNonVarargSpreadOnGenericCalls can be used to temporarily revert to the pre-1.8 behavior

Prohibit null-safety violation in lambdas passed to functions overloaded by lambda return type

Issue: KT-49658

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit returning null from lambdas passed to functions overloaded by those lambdas' return types when overloads don't allow a nullable return type. Previously, it was allowed when null was returned from one of the branches of the when operator

Deprecation cycle:

- 1.6.20: report a type mismatch warning (or an error in the progressive mode)
- 1.8.0: raise the warning to an error, -XXLanguage:-DontLoseDiagnosticsDuringOverloadResolutionByReturnType can be used to temporarily revert to the pre-1.8 behavior

Keep nullability when approximating local types in public signatures

Issue: KT-53982

Component: Core language

Incompatible change type: source, binary

Short summary: when a local or anonymous type is returned from an expression-body function without an explicitly specified return type, Kotlin compiler infers (or approximates) the return type using the known supertype of that type. During this, the compiler can infer a non-nullable type where the null value could in fact be returned

Deprecation cycle:

- 1.8.0: approximate flexible types by flexible supertypes
- 1.8.0: report a warning when a declaration is inferred to have a non-nullable type that should be nullable, prompting users to specify the type explicitly
- 1.9.0: approximate nullable types by nullable supertypes, -XXLanguage:-KeepNullabilityWhenApproximatingLocalType can be used to temporarily revert to the pre-1.9 behavior

Do not propagate deprecation through overrides

Issue: KT-47902

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will no longer propagate deprecation from a deprecated member in the superclass to its overriding member in the subclass, thus providing an explicit mechanism for deprecating a member of the superclass while leaving it non-deprecated in the subclass

Deprecation cycle:

- 1.6.20: reporting a warning with the message of the future behavior change and a prompt to either suppress this warning or explicitly write a @Deprecated annotation on an override of a deprecated member
- 1.9.0: stop propagating deprecation status to the overridden members. This change also takes effect immediately in the progressive mode

Prohibit implicit inferring a type variable into an upper bound in the builder inference context

Issue: KT-47986

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit inferring a type variable into the corresponding type parameter's upper bound in the absence of any use-site type information in the scope of builder inference lambda functions, the same way as it does currently in other contexts

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) when a type parameter is inferred into declared upper bounds in the absence of use-site type information
- 1.9.0: raise the warning to an error, -XXLanguage:-ForbidInferringPostponedTypeVariableIntoDeclaredUpperBound can be used to temporarily revert to the pre-1.9 behavior

Prohibit using collection literals in annotation classes anywhere except their parameters declaration

Issue: KT-39041

Component: Core language

Incompatible change type: source

Short summary: Kotlin allows using collection literals in a restricted way - for passing arrays to parameters of annotation classes or specifying default values for these parameters. However besides that, Kotlin allowed using collections literals anywhere else inside an annotation class, for example, in its nested object. Kotlin 1.9 will prohibit using collection literals in annotation classes anywhere except their parameters' default values.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on array literals in nested objects in annotation classes
- 1.9.0: raise the warning to an error

Prohibit forward referencing of parameters with default values in default value expressions

Issue: KT-25694

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit forward referencing of parameters with default values in default value expressions of other parameters. This ensures that by the time the parameter is accessed in a default value expression, it would already have a value either passed to the function or initialized by its own default value expression

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when a parameter with default value is references in default value of another parameter that comes before it
- 1.9.0: raise the warning to an error, -XXLanguage:-ProhibitIllegalValueParameterUsageInDefaultArguments can be used to temporarily revert to the pre-1.9 behavior

Prohibit extension calls on inline functional parameters

Issue: KT-52502

Component: Core language

Incompatible change type: source

Short summary: while Kotlin allowed passing an inline functional parameter to another inline function as a receiver, it always resulted in compiler exceptions when compiling such code. Kotlin 1.9 will prohibit this, thus reporting an error instead of crashing the compiler

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) for inline extension calls on inline functional parameters
- 1.9.0: raise the warning to an error

Prohibit calls to infix functions named suspend with an anonymous function argument

Issue: KT-49264

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will no longer allow calling infix functions named suspend that have a single argument of a functional type passed as an anonymous function literal

Deprecation cycle:

- 1.7.20: report a warning on suspend infix calls with an anonymous function literal
- 1.9.0: raise the warning to an error, -XXLanguage:-ModifierNonBuiltinSuspendFunError can be used to temporarily revert to the pre-1.9 behavior
- >=1.10: Change how the suspend fun token sequence is interpreted by the parser

Prohibit using captured type parameters in inner classes against their variance

Issue: KT-50947

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit using type parameters of an outer class having in or out variance in an inner class of that class in positions violating that type parameters' declared variance

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when an outer class' type parameter usage position violates the variance rules of that parameter
- 1.9.0: raise the warning to an error, -XXLanguage:-ReportTypeVarianceConflictOnQualifierArguments can be used to temporarily revert to the pre-1.9 behavior

Prohibit recursive call of a function without explicit return type in compound assignment operators

Issue: KT-48546

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit calling a function without explicitly specified return type in an argument of a compound assignment operator inside that function's body, as it currently does in other expressions inside the body of that function

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when a function without explicitly specified return type is called recursively in that function's body in a compound assignment operator argument
- 1.9.0: raise the warning to an error

Prohibit unsound calls with expected @NotNull T and given Kotlin generic parameter with nullable bound

Issue: KT-36770

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit method calls where a value of a potentially nullable generic type is passed for a @NotNull-annotated parameter of a Java method

Deprecation cycle:

- 1.5.20: report a warning when an unconstrained generic type parameter is passed where a non-nullable type is expected
- 1.9.0: report a type mismatch error instead of the warning above, -XXLanguage:-ProhibitUsingNullableTypeParameterAgainstNotNullAnnotated can be used to temporarily revert to the pre-1.8 behavior

Prohibit access to members of a companion of an enum class from entry initializers of this enum

Issue: KT-49110

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit all kinds of access to the companion object of an enum from an enum entry initializer

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) on such companion member access
- 1.9.0: raise the warning to an error, -XXLanguage:-ProhibitAccessToEnumCompanionMembersInEnumConstructorCall can be used to temporarily revert to the pre-1.8 behavior

Deprecate and remove Enum.declaringClass synthetic property

Issue: KT-49653

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin allowed using the synthetic property declaringClass on Enum values produced from the method getDeclaringClass() of the underlying Java class java.lang.Enum even though this method is not available for Kotlin Enum type. Kotlin 1.9 will prohibit using this property, proposing to migrate to the extension property declaringJavaClass instead

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on declaringClass property usages, propose the migration to declaringJavaClass extension
- 1.9.0: raise the warning to an error, -XXLanguage:-ProhibitEnumDeclaringClass can be used to temporarily revert to the pre-1.9 behavior
- >=1.10: remove declaringClass synthetic property

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

Issue: KT-46329

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6.20 warns about the usage of the enable and compatibility modes of the -Xjvm-default compiler option

Deprecation cycle:

- 1.6.20: introduce a warning on the enable and compatibility modes of the -Xjvm-default compiler option
- >= 1.9: raise this warning to an error

Standard library

Warn about potential overload resolution change when Range/Progression starts implementing Collection

Issue: KT-49276

Component: Core language / kotlin-stdlib

Incompatible change type: source

Short summary: it is planned to implement the Collection interface in the standard progressions and concrete ranges inherited from them in Kotlin 1.9. This could make a different overload selected in the overload resolution if there are two overloads of some method, one accepting an element and another accepting a collection. Kotlin will make this situation visible by reporting a warning or an error when such overloaded method is called with a range or progression argument

Deprecation cycle:

- 1.6.20: report a warning when an overloaded method is called with the standard progression or its range inheritor as an argument if implementing the Collection interface by this progression/range leads to another overload being selected in this call in future
- 1.8.0: raise this warning to an error
- 1.9.0: stop reporting the error, implement Collection interface in progressions thus changing the overload resolution result in the affected cases

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

> Issue: KT-39330
>
> Component: kotlin-stdlib (JS)
>
> Incompatible change type: source
>
> Short summary: declarations from the kotlin.dom and kotlin.browser packages are moved to the corresponding kotlinx.* packages to prepare for extracting them from stdlib
>
> Deprecation cycle:
>
> - 1.4.0: introduce the replacement API in kotlinx.dom and kotlinx.browser packages
> - 1.4.0: deprecate the API in kotlin.dom and kotlin.browser packages and propose the new API above as a replacement
> - 1.6.0: raise the deprecation level to an error
> - 1.8.20: remove the deprecated functions from stdlib for JS-IR target
> - >= 1.9: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

> Issue: KT-48587
>
> Component: kotlin-stdlib (JS)
>
> Incompatible change type: source
>
> Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: String.concat(String), String.match(regex: String), String.matches(regex: String), and the sort functions on arrays taking a comparison function, for example, Array<out T>.sort(comparison: (a: T, b: T) -> Int)
>
> Deprecation cycle:
>
> - 1.6.0: deprecate the affected functions with a warning
> - 1.9.0: raise the deprecation level to an error
> - >=1.10.0: remove the deprecated functions from the public API

Tools

Raise deprecation level of classpath property of KotlinCompile task

Issue: KT-51679

Component: Gradle

Incompatible change type: source

Short summary: the classpath property of the KotlinCompile task is deprecated

Deprecation cycle:

- 1.7.0: the classpath property is deprecated
- 1.8.0: raise the deprecation level to an error
- >=1.9.0: remove the deprecated functions from the public API

Remove kapt.use.worker.api Gradle property

Issue: KT-48827

Component: Gradle

Incompatible change type: behavioral

Short summary: remove the kapt.use.worker.api property that allowed to run kapt via Gradle Workers API (default: true)

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- 1.8.0: remove this property

Remove kotlin.compiler.execution.strategy system property

Issue: KT-51831

Component: Gradle

Incompatible change type: behavioral

Short summary: remove the kotlin.compiler.execution.strategy system property used to choose a compiler execution strategy. Use the Gradle property kotlin.compiler.execution.strategy or the compile task property compilerExecutionStrategy instead

Deprecation cycle:

- 1.7.0: raise the deprecation level to a warning
- 1.8.0: remove the property

Changes in compiler options

Issues: KT-27301, KT-48532

Component: Gradle

Incompatible change type: source, binary

Short summary: this change might affect Gradle plugins authors. In kotlin-gradle-plugin, there are additional generic parameters to some internal types (you should add generic types or *). KotlinNativeLink task does not inherit the AbstractKotlinNativeCompile task anymore. KotlinJsCompilerOptions.outputFile and the related KotlinJsOptions.outputFile options are deprecated. Use the Kotlin2JsCompile.outputFileProperty task input instead. The kotlinOptions task input and the kotlinOptions{...} task DSL are in a support mode and will be deprecated in upcoming releases. compilerOptions and kotlinOptions can not be changed on a task execution phase (see one exception in What's new in Kotlin 1.8). freeCompilerArgs returns an immutable List<String> – kotlinOptions.freeCompilerArgs.remove("something") will fail. The useOldBackend property that allowed to use the old JVM backend is removed

Deprecation cycle:

- 1.8.0: KotlinNativeLink task does not inherit the AbstractKotlinNativeCompile. KotlinJsCompilerOptions.outputFile and the related KotlinJsOptions.outputFile options are deprecated. The useOldBackend property that allowed to use the old JVM backend is removed.

Deprecate kotlin.internal.single.build.metrics.file property

Issue: KT-53357

Component: Gradle

Incompatible change type: source

Short summary: deprecate the kotlin.internal.single.build.metrics.file property used to define a single file for build reports. Use the property kotlin.build.report.single_file instead with kotlin.build.report.output=single_file

Deprecation cycle:

- 1.8.0: raise the deprecation level to a warning >= 1.9: delete the property

Compatibility guide for Kotlin 1.7.20

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

Usually incompatible changes happen only in feature releases, but this time we have to introduce two such changes in an incremental release to limit spread of the problems introduced by changes in Kotlin 1.7.

This document summarizes them, providing a reference for migration from Kotlin 1.7.0 and 1.7.10 to Kotlin 1.7.20.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Rollback attempt to fix proper constraints processing

> Issue: KT-53813
>
> Component: Core language
>
> Incompatible change type: source
>
> Short summary: Rollback an attempt of fixing issues in type inference constraints processing appeared in 1.7.0 after implementing the change described in KT-52668. The attempt was made in 1.7.10, but it in turn introduced new problems.
>
> Deprecation cycle:
>
> - 1.7.20: Rollback to 1.7.0 behavior

Forbid some builder inference cases to avoid problematic interaction with multiple lambdas and resolution

> Issue: KT-53797
>
> Component: Core language
>
> Incompatible change type: source
>
> Short summary: Kotlin 1.7 introduced a feature called unrestricted builder inference, so that even the lambdas passed to parameters not annotated with @BuilderInference could benefit from the builder inference. However, that could cause several problems if more than one such lambda occurred in a function invocation.
>
> Kotlin 1.7.20 will report an error if more than one lambda function having the corresponding parameter not annotated with @BuilderInference requires using builder inference to complete inferring the types in the lambda.
>
> Deprecation cycle:
>
> - 1.7.20: report an error on such lambda functions,
>   -XXLanguage:+NoBuilderInferenceWithoutAnnotationRestriction can be used to temporarily revert to the pre-1.7.20 behavior

Compatibility guide for Kotlin 1.7

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.6 to Kotlin 1.7.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Make safe call result always nullable

Issue: KT-46860

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will consider the type of safe call result always nullable, even when the receiver of the safe call is non-nullable

Deprecation cycle:

- <1.3: report a warning on an unnecessary safe call on non-nullable receivers
- 1.6.20: warn additionally that the result of an unnecessary safe call will change its type in the next version
- 1.7.0: change the type of safe call result to nullable,
  -XXLanguage:-SafeCallsAreAlwaysNullable can be used to temporarily revert to the pre-1.7 behavior

Prohibit the delegation of super calls to an abstract superclass member

Issues: KT-45508, KT-49017, KT-38078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will report a compile error when an explicit or implicit super call is delegated to an abstract member of the superclass, even if there's a default implementation in a super interface

Deprecation cycle:

- 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
- 1.7.0: report an error if a super call, in fact, accesses an abstract member from a superclass
- 1.7.0: report an error if the -Xjvm-default=all or -Xjvm-default=all-compatibility compatibility modes are enabled; report an error in the progressive mode
- >=1.8.0: report an error in all cases

Prohibit exposing non-public types through public properties declared in a non-public primary constructor

Issue: KT-28078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prevent declaring public properties having non-public types in a private primary constructor. Accessing such properties from another package could lead to an IllegalAccessError

Deprecation cycle:

- 1.3.20: report a warning on a public property that has a non-public type and is declared in a non-public constructor
- 1.6.20: raise this warning to an error in the progressive mode
- 1.7.0: raise this warning to an error

Prohibit access to uninitialized enum entries qualified with the enum name

Issue: KT-41124

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will prohibit access to uninitialized enum entries from the enum static initializer block when these entries are qualified with the enum name

Deprecation cycle:

- 1.7.0: report an error when uninitialized enum entries are accessed from the enum static initializer block

Prohibit computing constant values of complex boolean expressions in when condition branches and conditions of loops

Issue: KT-39883

Component: Core language

Incompatible change type: source

Short summary: Kotlin will no longer make exhaustiveness and control flow assumptions based on constant boolean expressions other than literal true and false

Deprecation cycle:

- 1.5.30: report a warning when exhaustiveness of when or control flow reachability is determined based on a complex constant boolean expression in when branch or loop condition
- 1.7.0: raise this warning to an error

Make when statements with enum, sealed, and Boolean subjects exhaustive by default

Issue: KT-47709

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will report an error about the when statement with an enum, sealed, or Boolean subject being non-exhaustive

Deprecation cycle:

- 1.6.0: introduce a warning when the when statement with an enum, sealed, or Boolean subject is non-exhaustive (error in the progressive mode)
- 1.7.0: raise this warning to an error

Deprecate confusing grammar in when-with-subject

Issue: KT-48385

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 deprecated several confusing grammar constructs in when condition expressions

Deprecation cycle:

- 1.6.20: introduce a deprecation warning on the affected expressions
- 1.8.0: raise this warning to an error
- >= 1.8: repurpose some deprecated constructs for new language features

Type nullability enhancement improvements

Issue: KT-48623

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.7 will change how it loads and interprets type nullability annotations in Java code

Deprecation cycle:

- 1.4.30: introduce warnings for cases where more precise type nullability could lead to an error
- 1.7.0: infer more precise nullability of Java types, -XXLanguage:-TypeEnhancementImprovementsInStrictMode can be used to temporarily revert to the pre-1.7 behavior

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type was needed semantically

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, -Xuse-old-backend can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 1.7.20: fix the downcast behavior in other affected cases

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

Issue: KT-46329

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6.20 warns about the usage of enable and compatibility modes of the -Xjvm-default compiler option

Deprecation cycle:

- 1.6.20: introduce a warning on the enable and compatibility modes of the -Xjvm-default compiler option
- >= 1.8.0: raise this warning to an error

Prohibit calls to functions named suspend with a trailing lambda

Issue: KT-22562

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 no longer allows calling user functions named suspend that have the single argument of a functional type passed as a trailing lambda

Deprecation cycle:

- 1.3.0: introduce a warning on such function calls
- 1.6.0: raise this warning to an error
- 1.7.0: introduce changes to the language grammar so that suspend before { is parsed as a keyword

Prohibit smart cast on a base class property if the base class is from another module

Issue: KT-52629

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will no longer allow smart casts on properties of a superclass if that class is located in another module

Deprecation cycle:

- 1.6.0: report a warning on a smart cast on a property declared in the superclass located in another module
- 1.7.0: raise this warning to an error,
  -XXLanguage:-ProhibitSmartcastsOnPropertyFromAlienBaseClass can be used to temporarily revert to the pre-1.7 behavior

Do not neglect meaningful constraints during type inference

Issue: KT-52668

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4–1.6 neglected some type constraints during type inference due to an incorrect optimization. It could allow writing unsound code, causing ClassCastException at runtime. Kotlin 1.7 takes these constraints into account, thus prohibiting the unsound code

Deprecation cycle:

- 1.5.20: report a warning on expressions where a type mismatch would happen if all the type inference constraints were taken into account
- 1.7.0: take all the constraints into account, thus raising this warning to an error,
  -XXLanguage:-ProperTypeInferenceConstraintsProcessing can be used to temporarily revert to the pre-1.7 behavior

Standard library

Gradually change the return type of collection min and max functions to non-nullable

Issue: KT-38854

Component: kotlin-stdlib

Incompatible change type: source

Short summary: the return type of collection min and max functions will be changed to non-nullable in Kotlin 1.7

Deprecation cycle:

- 1.4.0: introduce ...OrNull functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.0: raise the deprecation level of the affected API to an error
- 1.6.0: hide the deprecated functions from the public API
- 1.7.0: reintroduce the affected API but with non-nullable return type

Deprecate floating-point array functions: contains, indexOf, lastIndexOf

Issue: KT-28753

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Kotlin deprecates floating-point array functions contains, indexOf, lastIndexOf that compare values using the IEEE-754 order instead of the total order

Deprecation cycle:

- 1.4.0: deprecate the affected functions with a warning
- 1.6.0: raise the deprecation level to an error
- 1.7.0: hide the deprecated functions from the public API

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the kotlin.dom and kotlin.browser packages are moved to the corresponding kotlinx.* packages to prepare for extracting them from stdlib

Deprecation cycle:

- 1.4.0: introduce the replacement API in kotlinx.dom and kotlinx.browser packages
- 1.4.0: deprecate the API in kotlin.dom and kotlin.browser packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- >= 1.8: remove the deprecated functions from stdlib
- >= 1.8: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: String.concat(String), String.match(regex: String), String.matches(regex: String), and the sort functions on arrays taking a comparison function, for example, Array<out T>.sort(comparison: (a: T, b: T) -> Int)

Deprecation cycle:

- 1.6.0: deprecate the affected functions with a warning
- 1.8.0: raise the deprecation level to an error
- 1.9.0: remove the deprecated functions from the public API

Tools

Remove KotlinGradleSubplugin class

Issue: KT-48831

Component: Gradle

Incompatible change type: source

Short summary: remove the KotlinGradleSubplugin class. Use the KotlinCompilerPluginSupportPlugin class instead

Deprecation cycle:

- 1.6.0: raise the deprecation level to an error
- 1.7.0: remove the deprecated class

Remove useIR compiler option

Issue: KT-48847

Component: Gradle

Incompatible change type: source

Short summary: remove the deprecated and hidden useIR compiler option

Deprecation cycle:

- 1.5.0: raise the deprecation level to a warning
- 1.6.0: hide the option
- 1.7.0: remove the deprecated option

Deprecate kapt.use.worker.api Gradle property

Issue: KT-48826

Component: Gradle

Incompatible change type: source

Short summary: deprecate the kapt.use.worker.api property that allowed to run kapt via Gradle Workers API (default: true)

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- >= 1.8.0: remove this property

Remove kotlin.experimental.coroutines Gradle DSL option and kotlin.coroutines Gradle property

Issue: KT-50494

Component: Gradle

Incompatible change type: source

Short summary: remove the kotlin.experimental.coroutines Gradle DSL option and the kotlin.coroutines property

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- 1.7.0: remove the DSL option, its enclosing experimental block, and the property

Deprecate useExperimentalAnnotation compiler option

Issue: KT-47763

Component: Gradle

Incompatible change type: source

Short summary: remove the hidden useExperimentalAnnotation() Gradle function used to opt in to using an API in a module. optIn() function can be used instead

Deprecation cycle:

- 1.6.0: hide the deprecation option
- 1.7.0: remove the deprecated option

Deprecate kotlin.compiler.execution.strategy system property

Issue: KT-51830

Component: Gradle

Incompatible change type: source

Short summary: deprecate the kotlin.compiler.execution.strategy system property used to choose a compiler execution strategy. Use the Gradle property kotlin.compiler.execution.strategy or the compile task property compilerExecutionStrategy instead

Deprecation cycle:

- 1.7.0: raise the deprecation level to a warning
- > 1.7.0: remove the property

Remove kotlinOptions.jdkHome compiler option

Issue: KT-46541

Component: Gradle

Incompatible change type: source

Short summary: remove the kotlinOptions.jdkHome compiler option used to include a custom JDK from the specified location into the classpath instead of the default JAVA_HOME. Use Java toolchains instead

Deprecation cycle:

- 1.5.30: raise the deprecation level to a warning
- > 1.7.0: remove the option

Remove noStdlib compiler option

Issue: KT-49011

Component: Gradle

Incompatible change type: source

Short summary: remove the noStdlib compiler option. The Gradle plugin uses the kotlin.stdlib.default.dependency=true property to control whether the Kotlin standard library is present

Deprecation cycle:

- 1.5.0: raise the deprecation level to a warning
- 1.7.0: remove the option

Remove kotlin2js and kotlin-dce-plugin plugins

Issue: KT-48276

Component: Gradle

Incompatible change type: source

Short summary: remove the kotlin2js and kotlin-dce-plugin plugins. Instead of kotlin2js, use the new org.jetbrains.kotlin.js plugin. Dead code elimination (DCE) works when the Kotlin/JS Gradle plugin is properly configured

Deprecation cycle:

- 1.4.0: raise the deprecation level to a warning
- 1.7.0: remove the plugins

Changes in compile tasks

> Issue: KT-32805
>
> Component: Gradle
>
> Incompatible change type: source
>
> Short summary: Kotlin compile tasks no longer inherit the Gradle AbstractCompile task and that's why the sourceCompatibility and targetCompatibility inputs are no longer available in Kotlin users' scripts. The SourceTask.stableSources input is no longer available. The sourceFilesExtensions input was removed. The deprecated Gradle destinationDir: File output was replaced with the destinationDirectory: DirectoryProperty output. The classpath property of the KotlinCompile task is deprecated
>
> Deprecation cycle:
>
> - 1.7.0: inputs are not available, the output is replaced, the classpath property is deprecated

Compatibility guide for Kotlin 1.6

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.5 to Kotlin 1.6.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Make when statements with enum, sealed, and Boolean subjects exhaustive by default

> Issue: KT-47709
>
> Component: Core language
>
> Incompatible change type: source
>
> Short summary: Kotlin 1.6 will warn about the when statement with an enum, sealed, or Boolean subject being non-exhaustive
>
> Deprecation cycle:
>
> - 1.6.0: introduce a warning when the when statement with an enum, sealed, or Boolean subject is non-exhaustive (error in the progressive mode)
> - 1.7.0: raise this warning to an error

Deprecate confusing grammar in when-with-subject

Issue: KT-48385

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will deprecate several confusing grammar constructs in when condition expressions

Deprecation cycle:

- 1.6.20: introduce a deprecation warning on the affected expressions
- 1.8.0: raise this warning to an error
- >= 1.8: repurpose some deprecated constructs for new language features

Prohibit access to class members in the super constructor call of its companion and nested objects

Issue: KT-25289

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will report an error for arguments of super constructor call of companion and regular objects if the receiver of such arguments refers to the containing declaration

Deprecation cycle:

- 1.5.20: introduce a warning on the problematic arguments
- 1.6.0: raise this warning to an error, -XXLanguage:-ProhibitSelfCallsInNestedObjects can be used to temporarily revert to the pre-1.6 behavior

Type nullability enhancement improvements

Issue: KT-48623

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.7 will change how it loads and interprets type nullability annotations in Java code

Deprecation cycle:

- 1.4.30: introduce warnings for cases where more precise type nullability could lead to an error
- 1.7.0: infer more precise nullability of Java types, -XXLanguage:-TypeEnhancementImprovementsInStrictMode can be used to temporarily revert to the pre-1.7 behavior

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type was needed semantically

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, -Xuse-old-backend can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 1.6.20: fix the downcast behavior in other affected cases

Prohibit declarations of repeatable annotation classes whose container annotation violates JLS

Issue: KT-47928

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6 will check that the container annotation of a repeatable annotation satisfies the same requirements as in JLS 9.6.3: array-typed value method, retention, and target

Deprecation cycle:

- 1.5.30: introduce a warning on repeatable container annotation declarations violating JLS requirements (error in the progressive mode)
- 1.6.0: raise this warning to an error, -XXLanguage:-RepeatableAnnotationContainerConstraints can be used to temporarily disable the error reporting

Prohibit declaring a nested class named Container in a repeatable annotation class

Issue: KT-47971

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6 will check that a repeatable annotation declared in Kotlin doesn't have a nested class with the predefined name Container

Deprecation cycle:

- 1.5.30: introduce a warning on nested classes with the name Container in a Kotlin-repeatable annotation class (error in the progressive mode)
- 1.6.0: raise this warning to an error, -XXLanguage:-RepeatableAnnotationContainerConstraints can be used to temporarily disable the error reporting

Prohibit @JvmField on a property in the primary constructor that overrides an interface property

Issue: KT-32753

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw annotating a property declared in the primary constructor that overrides an interface property with the @JvmField annotation

Deprecation cycle:

- 1.5.20: introduce a warning on the @JvmField annotation on such properties in the primary constructor
- 1.6.0: raise this warning to an error, -XXLanguage:-ProhibitJvmFieldOnOverrideFromInterfaceInPrimaryConstructor can be used to temporarily disable the error reporting

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

Issue: KT-46329

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6.20 will warn about the usage of enable and compatibility modes of the -Xjvm-default compiler option

Deprecation cycle:

- 1.6.20: introduce a warning on the enable and compatibility modes of the -Xjvm-default compiler option
- >= 1.8.0: raise this warning to an error

Prohibit super calls from public-abi inline functions

Issue: KT-45379

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw calling functions with a super qualifier from public or protected inline functions and properties

Deprecation cycle:

- 1.5.0: introduce a warning on super calls from public or protected inline functions or property accessors
- 1.6.0: raise this warning to an error, -XXLanguage:-ProhibitSuperCallsFromPublicInline can be used to temporarily disable the error reporting

Prohibit protected constructor calls from public inline functions

Issue: KT-48860

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw calling protected constructors from public or protected inline functions and properties

Deprecation cycle:

- 1.4.30: introduce a warning on protected constructor calls from public or protected inline functions or property accessors
- 1.6.0: raise this warning to an error, -XXLanguage:-ProhibitProtectedConstructorCallFromPublicInline can be used to temporarily disable the error reporting

Prohibit exposing private nested types from private-in-file types

Issue: KT-20094

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw exposing private nested types and inner classes from private-in-file types

Deprecation cycle:

- 1.5.0: introduce a warning on private types exposed from private-in-file types
- 1.6.0: raise this warning to an error, -XXLanguage:-PrivateInFileEffectiveVisibility can be used to temporarily disable the error reporting

Annotation target is not analyzed in several cases for annotations on a type

Issue: KT-28449

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will no longer allow annotations on types that should not be applicable to types

Deprecation cycle:

- 1.5.20: introduce an error in the progressive mode
- 1.6.0: introduce an error, -XXLanguage:-ProperCheckAnnotationsTargetInTypeUsePositions can be used to temporarily disable the error reporting

Prohibit calls to functions named suspend with a trailing lambda

Issue: KT-22562

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will no longer allow calling functions named suspend that have the single argument of a functional type passed as a trailing lambda

Deprecation cycle:

- 1.3.0: introduce a warning on such function calls
- 1.6.0: raise this warning to an error
- >= 1.7.0: introduce changes to the language grammar, so that suspend before { is parsed as a keyword

Standard library

Remove brittle contains optimization in minus/removeAll/retainAll

Issue: KT-45438

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Kotlin 1.6 will no longer perform conversion to set for the argument of functions and operators that remove several elements from collection/iterable/array/sequence.

Deprecation cycle:

- < 1.6: the old behavior: the argument is converted to set in some cases
- 1.6.0: if the function argument is a collection, it's no longer converted to Set. If it's not a collection, it can be converted to List instead.
  The old behavior can be temporarily turned back on JVM by setting the system property kotlin.collections.convert_arg_to_set_in_removeAll=true
- >= 1.7: the system property above will no longer have an effect

Change value generation algorithm in Random.nextLong

Issue: KT-47304

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Kotlin 1.6 changes the value generation algorithm in the Random.nextLong function to avoid producing values out of the specified range.

Deprecation cycle:

- 1.6.0: the behavior is fixed immediately

Gradually change the return type of collection min and max functions to non-nullable

Issue: KT-38854

Component: kotlin-stdlib

Incompatible change type: source

Short summary: the return type of collection min and max functions will be changed to non-nullable in Kotlin 1.7

Deprecation cycle:

- 1.4.0: introduce ...OrNull functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.0: raise the deprecation level of the affected API to an error
- 1.6.0: hide the deprecated functions from the public API
- >= 1.7: reintroduce the affected API but with non-nullable return type

Deprecate floating-point array functions: contains, indexOf, lastIndexOf

Issue: KT-28753

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Kotlin deprecates floating-point array functions contains, indexOf, lastIndexOf that compare values using the IEEE-754 order instead of the total order

Deprecation cycle:

- 1.4.0: deprecate the affected functions with a warning
- 1.6.0: raise the deprecation level to an error
- >= 1.7: hide the deprecated functions from the public API

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the kotlin.dom and kotlin.browser packages are moved to the corresponding kotlinx.* packages to prepare for extracting them from stdlib

Deprecation cycle:

- 1.4.0: introduce the replacement API in kotlinx.dom and kotlinx.browser packages
- 1.4.0: deprecate the API in kotlin.dom and kotlin.browser packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- >= 1.7: remove the deprecated functions from stdlib
- >= 1.7: move the API in kotlinx.* packages to a separate library

Make Regex.replace function not inline in Kotlin/JS

Issue: KT-27738

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: the Regex.replace function with the functional transform parameter will no longer be inline in Kotlin/JS

Deprecation cycle:

- 1.6.0: remove the inline modifier from the affected function

Different behavior of the Regex.replace function in JVM and JS when replacement string contains group reference

Issue: KT-28378

Component: kotlin-stdlib (JS)

Incompatible change type: behavioral

Short summary: the function Regex.replace in Kotlin/JS with the replacement pattern string will follow the same syntax of that pattern as in Kotlin/JVM

Deprecation cycle:

- 1.6.0: change the replacement pattern handling in Regex.replace of the Kotlin/JS stdlib

Use the Unicode case folding in JS Regex

Issue: KT-45928

Component: kotlin-stdlib (JS)

Incompatible change type: behavioral

Short summary: the Regex class in Kotlin/JS will use unicode flag when calling the underlying JS Regular expressions engine to search and compare characters according to the Unicode rules. This brings certain version requirements of the JS environment and causes more strict validation of unnecessary escaping in the regex pattern string.

Deprecation cycle:

- 1.5.0: enable the Unicode case folding in most functions of the JS Regex class
- 1.6.0: enable the Unicode case folding in the Regex.replaceFirst function

Deprecate some JS-only API

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: String.concat(String), String.match(regex: String), String.matches(regex: String), and the sort functions on arrays taking a comparison function, for example, Array<out T>.sort(comparison: (a: T, b: T) -> Int)

Deprecation cycle:

- 1.6.0: deprecate the affected functions with a warning
- 1.7.0: raise the deprecation level to an error
- 1.8.0: remove the deprecated functions from the public API

Hide implementation- and interop-specific functions from the public API of classes in Kotlin/JS

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source, binary

Short summary: the functions HashMap.createEntrySet and AbstactMutableCollection.toJSON change their visibility to internal

Deprecation cycle:

- 1.6.0: make the functions internal, thus removing them from the public API

Tools

Deprecate KotlinGradleSubplugin class

Issue: KT-48830

Component: Gradle

Incompatible change type: source

Short summary: the class KotlinGradleSubplugin will be deprecated in favor of KotlinCompilerPluginSupportPlugin

Deprecation cycle:

- 1.6.0: raise the deprecation level to an error
- >= 1.7.0: remove the deprecated class

Remove kotlin.useFallbackCompilerSearch build option

Issue: KT-46719

Component: Gradle

Incompatible change type: source

Short summary: remove the deprecated 'kotlin.useFallbackCompilerSearch' build option

Deprecation cycle:

- 1.5.0: raise the deprecation level to a warning
- 1.6.0: remove the deprecated option

Remove several compiler options

Issue: KT-48847

Component: Gradle

Incompatible change type: source

Short summary: remove the deprecated noReflect and includeRuntime compiler options

Deprecation cycle:

- 1.5.0: raise the deprecation level to an error
- 1.6.0: remove the deprecated options

Deprecate useIR compiler option

Issue: KT-48847

Component: Gradle

Incompatible change type: source

Short summary: hide the deprecated useIR compiler option

Deprecation cycle:

- 1.5.0: raise the deprecation level to a warning
- 1.6.0: hide the option
- >= 1.7.0: remove the deprecated option

Deprecate kapt.use.worker.api Gradle property

Issue: KT-48826

Component: Gradle

Incompatible change type: source

Short summary: deprecate the kapt.use.worker.api property that allowed to run kapt via Gradle Workers API (default: true)

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- >= 1.8.0: remove this property

Remove kotlin.parallel.tasks.in.project Gradle property

Issue: KT-46406

Component: Gradle

Incompatible change type: source

Short summary: remove the kotlin.parallel.tasks.in.project property

Deprecation cycle:

- 1.5.20: raise the deprecation level to a warning
- 1.6.20: remove this property

Deprecate kotlin.experimental.coroutines Gradle DSL option and kotlin.coroutines Gradle property

Issue: KT-50369

Component: Gradle

Incompatible change type: source

Short summary: deprecate the kotlin.experimental.coroutines Gradle DSL option and the kotlin.coroutines property

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- >= 1.7.0: remove the DSL option and the property

Compatibility guide for Kotlin 1.5

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.4 to Kotlin 1.5.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language and stdlib

Forbid spread operator in signature-polymorphic calls

> Issue: KT-35226
>
> Component: Core language
>
> Incompatible change type: source
>
> Short summary: Kotlin 1.5 will outlaw the use of spread operator (*) on signature-polymorphic calls
>
> Deprecation cycle:
>
> - < 1.5: introduce warning for the problematic operator at call-site
> - >= 1.5: raise this warning to an error, -XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall can be used to temporarily revert to pre-1.5 behavior

Forbid non-abstract classes containing abstract members invisible from that classes (internal/package-private)

> Issue: KT-27825
>
> Component: Core language
>
> Incompatible change type: source
>
> Short summary: Kotlin 1.5 will outlaw non-abstract classes containing abstract members invisible from that classes (internal/package-private)
>
> Deprecation cycle:
>
> - < 1.5: introduce warning for the problematic classes
> - >= 1.5: raise this warning to an error, -XXLanguage:-ProhibitInvisibleAbstractMethodsInSuperclasses can be used to temporarily revert to pre-1.5 behavior

Forbid using array based on non-reified type parameters as reified type arguments on JVM

Issue: KT-31227

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw using array based on non-reified type parameters as reified type arguments on JVM

Deprecation cycle:

- < 1.5: introduce warning for the problematic calls
- >= 1.5: raise this warning to an error, -XXLanguage:-ProhibitNonReifiedArraysAsReifiedTypeArguments can be used to temporarily revert to pre-1.5 behavior

Forbid secondary enum class constructors which do not delegate to the primary constructor

Issue: KT-35870

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw secondary enum class constructors which do not delegate to the primary constructor

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, -XXLanguage:-RequiredPrimaryConstructorDelegationCallInEnums can be used to temporarily revert to pre-1.5 behavior

Forbid exposing anonymous types from private inline functions

Issue: KT-33917

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw exposing anonymous types from private inline functions

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, -XXLanguage:-ApproximateAnonymousReturnTypesInPrivateInlineFunctions can be used to temporarily revert to pre-1.5 behavior

Forbid passing non-spread arrays after arguments with SAM-conversion

Issue: KT-35224

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw passing non-spread arrays after arguments with SAM-conversion

Deprecation cycle:

- 1.3.70: introduce warning for the problematic calls
- >= 1.5: raise this warning to an error, -XXLanguage:-ProhibitVarargAsArrayAfterSamArgument can be used to temporarily revert to pre-1.5 behavior

Support special semantics for underscore-named catch block parameters

Issue: KT-31567

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw references to the underscore symbol (_) that is used to omit parameter name of an exception in the catch block

Deprecation cycle:

- 1.4.20: introduce warning for the problematic references
- >= 1.5: raise this warning to an error, -XXLanguage:-ForbidReferencingToUnderscoreNamedParameterOfCatchBlock can be used to temporarily revert to pre-1.5 behavior

Change implementation strategy of SAM conversion from anonymous class-based to invokedynamic

Issue: KT-44912

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, implementation strategy of SAM (single abstract method) conversion will be changed from generating an anonymous class to using the invokedynamic JVM instruction

Deprecation cycle:

- 1.5: change implementation strategy of SAM conversion, -Xsam-conversions=class can be used to revert implementation scheme to the one that used before

Performance issues with the JVM IR-based backend

Issue: KT-48233

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 uses the IR-based backend for the Kotlin/JVM compiler by default. The old backend is still used by default for earlier language versions.

You might encounter some performance degradation issues using the new compiler in Kotlin 1.5. We are working on fixing such cases.

Deprecation cycle:

- < 1.5: by default, the old JVM backend is used
- >= 1.5: by default, the IR-based backend is used. If you need to use the old backend in Kotlin 1.5, add the following lines to the project's configuration file to temporarily revert to pre-1.5 behavior:

In Gradle:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useOldBackend = true
}
```

In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

Support for this flag will be removed in one of the future releases.

New field sorting in the JVM IR-based backend

Issue: KT-46378

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since version 1.5, Kotlin uses the IR-based backend that sorts JVM bytecode differently: it generates fields declared in the constructor before fields declared in the body, while it's vice versa for the old backend. The new sorting may change the behavior of programs that use serialization frameworks that depend on the field order, such as Java serialization.

Deprecation cycle:

- < 1.5: by default, the old JVM backend is used. It has fields declared in the body before fields declared in the constructor.
- >= 1.5: by default, the new IR-based backend is used. Fields declared in the constructor are generated before fields declared in the body. As a workaround, you can temporarily switch to the old backend in Kotlin 1.5. To do that, add the following lines to the project's configuration file:

In Gradle:

Kotlin

```kotlin
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

Groovy

```groovy
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useOldBackend = true
}
```

In Maven:

```xml
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

Support for this flag will be removed in one of the future releases.

Generate nullability assertion for delegated properties with a generic call in the delegate expression

Issue: KT-44304

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, the Kotlin compiler will emit nullability assertions for delegated properties with a generic call in the delegate expression

Deprecation cycle:

- 1.5: emit nullability assertion for delegated properties (see details in the issue), -Xuse-old-backend or -language-version 1.4 can be used to temporarily revert to pre-1.5 behavior

Turn warnings into errors for calls with type parameters annotated by @OnlyInputTypes

Issue: KT-45861

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw calls like contains, indexOf, and assertEquals with senseless arguments to improve type safety

Deprecation cycle:

- 1.4.0: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, -XXLanguage:-StrictOnlyInputTypesChecks can be used to temporarily revert to pre-1.5 behavior

Use the correct order of arguments execution in calls with named vararg

Issue: KT-17691

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will change the order of arguments execution in calls with named vararg

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, -XXLanguage:-UseCorrectExecutionOrderForVarargArguments can be used to temporarily revert to pre-1.5 behavior

Use default value of the parameter in operator functional calls

Issue: KT-42064

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will use default value of the parameter in operator calls

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- >= 1.5: behavior changed, -XXLanguage:-JvmIrEnabledByDefault can be used to temporarily revert to pre-1.5 behavior

Produce empty reversed progressions in for loops if regular progression is also empty

Issue: KT-42533

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will produce empty reversed progressions in for loops if regular progression is also empty

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- >= 1.5: behavior changed, -XXLanguage:-JvmIrEnabledByDefault can be used to temporarily revert to pre-1.5 behavior

Straighten Char-to-code and Char-to-digit conversions out

Issue: KT-23451

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Since Kotlin 1.5, conversions of Char to number types will be deprecated

Deprecation cycle:

- 1.5: deprecate Char.toInt()/toShort()/toLong()/toByte()/toDouble()/toFloat() and the reverse functions like Long.toChar(), and propose replacement

Inconsistent case-insensitive comparison of characters in kotlin.text functions

Issue: KT-45496

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, Char.equals will be improved in case-insensitive case by first comparing whether the uppercase variants of characters are equal, then whether the lowercase variants of those uppercase variants (as opposed to the characters themselves) are equal

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- 1.5: change behavior for Char.equals function

Remove default locale-sensitive case conversion API

Issue: KT-43023

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Since Kotlin 1.5, default locale-sensitive case conversion functions like String.toUpperCase() will be deprecated

Deprecation cycle:

- 1.5: deprecate case conversions functions with the default locale (see details in the issue), and propose replacement

Gradually change the return type of collection min and max functions to non-nullable

Issue: KT-38854

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: return type of collection min and max functions will be changed to non-nullable in 1.6

Deprecation cycle:

- 1.4: introduce ...OrNull functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.0: raise the deprecation level of the affected API to error
- >=1.6: reintroduce the affected API but with non-nullable return type

Raise the deprecation level of conversions of floating-point types to Short and Byte

Issue: KT-30360

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: conversions of floating-point types to Short and Byte deprecated in Kotlin 1.4 with WARNING level will cause errors since Kotlin 1.5.0.

Deprecation cycle:

- 1.4: deprecate Double.toShort()/toByte() and Float.toShort()/toByte() and propose replacement
- 1.5.0: raise the deprecation level to error

Tools

Do not mix several JVM variants of kotlin-test in a single project

Issue: KT-40225

Component: Gradle

Incompatible change type: behavioral

Short summary: several mutually exclusive kotlin-test variants for different testing frameworks could have been in a project if one of them is brought by a transitive dependency. From 1.5.0, Gradle won't allow having mutually exclusive kotlin-test variants for different testing frameworks.

Deprecation cycle:

- < 1.5: having several mutually exclusive kotlin-test variants for different testing frameworks is allowed
- >= 1.5: behavior changed,
  Gradle throws an exception like "Cannot select module with conflict on capability...". Possible solutions:
  - use the same kotlin-test variant and the corresponding testing framework as the transitive dependency brings.
  - find another variant of the dependency that doesn't bring the kotlin-test variant transitively, so you can use the testing framework you would like to use.
  - find another variant of the dependency that brings another kotlin-test variant transitively, which uses the same testing framework you would like to use.
  - exclude the testing framework that is brought transitively. The following example is for excluding JUnit 4:

```
configurations {
    testImplementation.get().exclude("org.jetbrains.kotlin", "kotlin-test-junit")
}
```

    After excluding the testing framework, test your application. If it stopped working, rollback excluding changes, use the same testing framework as the library does, and exclude your testing framework.

Compatibility guide for Kotlin 1.4

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.3 to Kotlin 1.4.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language and stdlib

Unexpected behavior with in infix operator and ConcurrentHashMap

Issue: KT-18053

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4 will outlaw auto operator contains coming from the implementors of java.util.Map written in Java

Deprecation cycle:

- < 1.4: introduce warning for problematic operators at call-site
- >= 1.4: raise this warning to an error, -XXLanguage:-ProhibitConcurrentHashMapContains can be used to temporarily revert to pre-1.4 behavior

Prohibit access to protected members inside public inline members

Issue: KT-21178

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4 will prohibit access to protected members from public inline members.

Deprecation cycle:

- < 1.4: introduce warning at call-site for problematic cases
- 1.4: raise this warning to an error, -XXLanguage:-ProhibitProtectedCallFromInline can be used to temporarily revert to pre-1.4 behavior

Contracts on calls with implicit receivers

Issue: KT-28672

Component: Core Language

Incompatible change type: behavioral

Short summary: smart casts from contracts will be available on calls with implicit receivers in 1.4

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-ContractsOnCallsWithImplicitReceiver can be used to temporarily revert to pre-1.4 behavior

Inconsistent behavior of floating-point number comparisons

Issues: KT-22723

Component: Core language

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, Kotlin compiler will use IEEE 754 standard to compare floating-point numbers

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-ProperIeee754Comparisons can be used to temporarily revert to pre-1.4 behavior

No smart cast on the last expression in a generic lambda

Issue: KT-15020

Component: Core Language

Incompatible change type: behavioral

Short summary: smart casts for last expressions in lambdas will be correctly applied since 1.4

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Do not depend on the order of lambda arguments to coerce result to Unit

Issue: KT-36045

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, lambda arguments will be resolved independently without implicit coercion to Unit

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Wrong common supertype between raw and integer literal type leads to unsound code

Issue: KT-35681

Components: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, common supertype between raw Comparable type and integer literal type will be more specific

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because several equal type variables are instantiated with a different types

Issue: KT-35679

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, Kotlin compiler will prohibit instantiating equal type variables with different types

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because of incorrect subtyping for intersection types

Issues: KT-22474

Component: Core language

Incompatible change type: source

Short summary: in Kotlin 1.4, subtyping for intersection types will be refined to work more correctly

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

No type mismatch with an empty when expression inside lambda

Issue: KT-17995

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, there will be a type mismatch for empty when expression if it's used as the last expression in a lambda

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Return type Any inferred for lambda with early return with integer literal in one of possible return values

Issue: KT-20226

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, integer type returning from a lambda will be more specific for cases when there is early return

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Proper capturing of star projections with recursive types

Issue: KT-33012

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, more candidates will become applicable because capturing for recursive types will work more correctly

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Common supertype calculation with non-proper type and flexible one leads to incorrect results

Issue: KT-37054

Component: Core language

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, common supertype between flexible types will be more specific protecting from runtime errors

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because of lack of captured conversion against nullable type argument

Issue: KT-35487

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, subtyping between captured and nullable types will be more correct protecting from runtime errors

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Preserve intersection type for covariant types after unchecked cast

Issue: KT-37280

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, uchecked casts of covariant types produce the intersection type for smart casts, not the type of the unchecked cast.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type variable leaks from builder inference because of using this expression

Issue: KT-32126

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, using this inside builder functions like sequence {} is prohibited if there are no other proper constraints

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Wrong overload resolution for contravariant types with nullable type arguments

Issue: KT-31670

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, if two overloads of a function that takes contravariant type arguments differ only by the nullability of the type (such as In<T> and In<T?>), the nullable type is considered more specific.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Builder inference with non-nested recursive constraints

Issue: KT-34975

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, builder functions such as sequence {} with type that depends on a recursive constraint inside the passed lambda cause a compiler error.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Eager type variable fixation leads to a contradictory constraint system

Issue: KT-25175

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, the type inference in certain cases works less eagerly allowing to find the constraint system that is not contradictory.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Prohibit tailrec modifier on open functions

Issue: KT-18541

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, functions can't have open and tailrec modifiers at the same time.

Deprecation cycle:

- < 1.4: report a warning on functions that have open and tailrec modifiers together (error in the progressive mode).
- >= 1.4: raise this warning to an error.

The INSTANCE field of a companion object more visible than the companion object class itself

Issue: KT-11567

Component: Kotlin/JVM

Incompatible change type: source

Short summary: since Kotlin 1.4, if a companion object is private, then its field INSTANCE will be also private

Deprecation cycle:

- < 1.4: the compiler generates object INSTANCE with a deprecated flag
- >= 1.4: companion object INSTANCE field has proper visibility

Outer finally block inserted before return is not excluded from thecatch interval of the inner try block without finally

Issue: KT-31923

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, the catch interval will be computed properly for nested try/catch blocks

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-ProperFinally can be used to temporarily revert to pre-1.4 behavior

Use the boxed version of an inline class in return type position for covariant and generic-specialized overrides

Issues: KT-30419

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, functions using covariant and generic-specialized overrides will return boxed values of inline classes

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Do not declare checked exceptions in JVM bytecode when using delegation to Kotlin interfaces

Issue: KT-35834

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will not generate checked exceptions during interface delegation to Kotlin interfaces

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-DoNotGenerateThrowsForDelegatedKotlinMembers can be used to temporarily revert to pre-1.4 behavior

Changed behavior of signature-polymorphic calls to methods with a single vararg parameter to avoid wrapping the argument into another array

Issue: KT-35469

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will not wrap the argument into another array on a signature-polymorphic call

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Incorrect generic signature in annotations when KClass is used as a generic parameter

Issue: KT-35207

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will fix incorrect type mapping in annotations when KClass is used as a generic parameter

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Forbid spread operator in signature-polymorphic calls

Issue: KT-35226

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will prohibit the use of spread operator (*) on signature-polymorphic calls

Deprecation cycle:

- < 1.4: report a warning on the use of a spread operator in signature-polymorphic calls
- >= 1.5: raise this warning to an error, -XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall can be used to temporarily revert to pre-1.4 behavior

Change initialization order of default values for tail-recursive optimized functions

Issue: KT-31540

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.4, the initialization order for tail-recursive functions will be the same as for regular functions

Deprecation cycle:

- < 1.4: report a warning at declaration-site for problematic functions
- >= 1.4: behavior changed, -XXLanguage:-ProperComputationOrderOfTailrecDefaultParameters can be used to temporarily revert to pre-1.4 behavior

Do not generate ConstantValue attribute for non-const vals

Issue: KT-16615

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.4, the compiler will not generate the ConstantValue attribute for non-const vals

Deprecation cycle:

- < 1.4: report a warning through an IntelliJ IDEA inspection
- >= 1.4: behavior changed, -XXLanguage:-NoConstantValueAttributeForNonConstVals can be used to temporarily revert to pre-1.4 behavior

Generated overloads for @JvmOverloads on open methods should be final

Issue: KT-33240

Components: Kotlin/JVM

Incompatible change type: source

Short summary: overloads for functions with @JvmOverloads will be generated as final

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-GenerateJvmOverloadsAsFinal can be used to temporarily revert to pre-1.4 behavior

Lambdas returning kotlin.Result now return boxed value instead of unboxed

Issue: KT-39198

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, lambdas returning values of kotlin.Result type will return boxed value instead of unboxed

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Unify exceptions from null checks

Issue: KT-22275

Component: Kotlin/JVM

Incompatible change type: behavior

Short summary: Starting from Kotlin 1.4, all runtime null checks will throw a java.lang.NullPointerException

Deprecation cycle:

- < 1.4: runtime null checks throw different exceptions, such as KotlinNullPointerException, IllegalStateException, IllegalArgumentException, and TypeCastException
- >= 1.4: all runtime null checks throw a java.lang.NullPointerException. -Xno-unified-null-checks can be used to temporarily revert to pre-1.4 behavior

Comparing floating-point values in array/list operations contains, indexOf, lastIndexOf: IEEE 754 or total order

Issue: KT-28753

Component: kotlin-stdlib (JVM)

Incompatible change type: behavioral

Short summary: the List implementation returned from Double/FloatArray.asList() will implement contains, indexOf, and lastIndexOf, so that they use total order equality

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Gradually change the return type of collection min and max functions to non-nullable

Issue: KT-38854

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: return type of collection min and max functions will be changed to non-nullable in 1.6

Deprecation cycle:

- 1.4: introduce ...OrNull functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.x: raise the deprecation level of the affected API to error
- >=1.6: reintroduce the affected API but with non-nullable return type

Deprecate appendln in favor of appendLine

Issue: KT-38754

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: StringBuilder.appendln() will be deprecated in favor of StringBuilder.appendLine()

Deprecation cycle:

- 1.4: introduce appendLine function as a replacement for appendln and deprecate appendln
- >=1.5: raise the deprecation level to error

Deprecate conversions of floating-point types to Short and Byte

Issue: KT-30360

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: since Kotlin 1.4, conversions of floating-point types to Short and Byte will be deprecated

Deprecation cycle:

- 1.4: deprecate Double.toShort()/toByte() and Float.toShort()/toByte() and propose replacement
- >=1.5: raise the deprecation level to error

Fail fast in Regex.findAll on an invalid startIndex

Issue: KT-28356

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, findAll will be improved to check that startIndex is in the range of the valid position indices of the input char sequence at the moment of entering findAll, and throw IndexOutOfBoundsException if it's not

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Remove deprecated kotlin.coroutines.experimental

Issue: KT-36083

Component: kotlin-stdlib

Incompatible change type: source

Short summary: since Kotlin 1.4, the deprecated kotlin.coroutines.experimental API is removed from stdlib

Deprecation cycle:

- < 1.4: kotlin.coroutines.experimental is deprecated with the ERROR level
- >= 1.4: kotlin.coroutines.experimental is removed from stdlib. On the JVM, a separate compatibility artifact is provided (see details in the issue).

Remove deprecated mod operator

Issue: KT-26654

Component: kotlin-stdlib

Incompatible change type: source

Short summary: since Kotlin 1.4, mod operator on numeric types is removed from stdlib

Deprecation cycle:

- < 1.4: mod is deprecated with the ERROR level
- >= 1.4: mod is removed from stdlib

Hide Throwable.addSuppressed member and prefer extension instead

Issue: KT-38777

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Throwable.addSuppressed() extension function is now preferred over the Throwable.addSuppressed() member function

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

capitalize should convert digraphs to title case

Issue: KT-38817

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: String.capitalize() function now capitalizes digraphs from the Serbo-Croatian Gaj's Latin alphabet in the title case (ǅ instead of Ǆ)

Deprecation cycle:

- < 1.4: digraphs are capitalized in the upper case (Ǆ)
- >= 1.4: digraphs are capitalized in the title case (ǅ)

Tools

Compiler arguments with delimiter characters must be passed in double quotes on Windows

Issue: KT-41309

Component: CLI

Incompatible change type: behavioral

Short summary: on Windows, kotlinc.bat arguments that contain delimiter characters (whitespace, =, ;, ,) now require double quotes (")

Deprecation cycle:

- < 1.4: all compiler arguments are passed without quotes
- >= 1.4: compiler arguments that contain delimiter characters (whitespace, =, ;, ,) require double quotes (")

KAPT: Names of synthetic $annotations() methods for properties have changed

Issue: KT-36926

Component: KAPT

Incompatible change type: behavioral

Short summary: names of synthetic $annotations() methods generated by KAPT for properties have changed in 1.4

Deprecation cycle:

- < 1.4: names of synthetic $annotations() methods for properties follow the template <propertyName>@annotations()
- >= 1.4: names of synthetic $annotations() methods for properties include the get prefix: get<PropertyName>@annotations()

Compatibility guide for Kotlin 1.3

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.2 to Kotlin 1.3.

Basic terms

In this document we introduce several kinds of compatibility:

- Source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- Binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- Behavioral: a change is said to be behavioral-incompatible if one and the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (e.g. from Java) is out of the scope of this document.

Incompatible changes

Evaluation order of constructor arguments regarding call

Issue: KT-19532

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: evaluation order with respect to class initialization is changed in 1.3

Deprecation cycle:

- <1.3: old behavior (see details in the Issue)
- >= 1.3: behavior changed, -Xnormalize-constructor-calls=disable can be used to temporarily revert to pre-1.3 behavior. Support for this flag is going to be removed in the next major release.

Missing getter-targeted annotations on annotation constructor parameters

Issue: KT-25287

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: getter-target annotations on annotations constructor parameters will be properly written to classfiles in 1.3

Deprecation cycle:

- <1.3: getter-target annotations on annotation constructor parameters are not applied
- >=1.3: getter-target annotations on annotation constructor parameters are properly applied and written to the generated code

Missing errors in class constructor's @get: annotations

Issue: KT-19628

Component: Core language

Incompatible change type: Source

Short summary: errors in getter-target annotations will be reported properly in 1.3

Deprecation cycle:

- <1.2: compilation errors in getter-target annotations were not reported, causing incorrect code to be compiled fine.
- 1.2.x: errors reported only by tooling, the compiler still compiles such code without any warnings
- >=1.3: errors reported by the compiler too, causing erroneous code to be rejected

Nullability assertions on access to Java types annotated with @NotNull

Issue: KT-20830

Component: Kotlin/JVM

Incompatible change type: Behavioral

Short summary: nullability assertions for Java-types annotated with not-null annotations will be generated more aggressively, causing code which passes null here to fail faster.

Deprecation cycle:

- <1.3: the compiler could miss such assertions when type inference was involved, allowing potential null propagation during compilation against binaries (see Issue for details).
- >=1.3: the compiler generates missed assertions. This can cause code which was (erroneously) passing nulls here fail faster. -XXLanguage:-StrictJavaNullabilityAssertions can be used to temporarily return to the pre-1.3 behavior. Support for this flag will be removed in the next major release.

Unsound smartcasts on enum members

Issue: KT-20772

Component: Core language

Incompatible change type: Source

Short summary: a smartcast on a member of one enum entry will be correctly applied to only this enum entry

Deprecation cycle:

- <1.3: a smartcast on a member of one enum entry could lead to an unsound smartcast on the same member of other enum entries.
- >=1.3: smartcast will be properly applied only to the member of one enum entry.
  -XXLanguage:-SoundSmartcastForEnumEntries will temporarily return old behavior. Support for this flag will be removed in the next major release.

val backing field reassignment in getter

Issue: KT-16681

Components: Core language

Incompatible change type: Source

Short summary: reassignment of the backing field of val-property in its getter is now prohibited

Deprecation cycle:

- <1.2: Kotlin compiler allowed to modify backing field of val in its getter. Not only it violates Kotlin semantic, but also generates ill-behaved JVM bytecode which reassigns final field.
- 1.2.X: deprecation warning is reported on code which reassigns backing field of val
- >=1.3: deprecation warnings are elevated to errors

Array capturing before the for-loop where it is iterated

Issue: KT-21354

Component: Kotlin/JVM

Incompatible change type: Source

Short summary: if an expression in for-loop range is a local variable updated in a loop body, this change affects loop execution. This is inconsistent with iterating over other containers, such as ranges, character sequences, and collections.

Deprecation cycle:

- <1.2: described code patterns are compiled fine, but updates to local variable affect loop execution
- 1.2.X: deprecation warning reported if a range expression in a for-loop is an array-typed local variable which is assigned in a loop body
- 1.3: change behavior in such cases to be consistent with other containers

Nested classifiers in enum entries

Issue: KT-16310

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, nested classifiers (classes, object, interfaces, annotation classes, enum classes) in enum entries are prohibited

Deprecation cycle:

- <1.2: nested classifiers in enum entries are compiled fine, but may fail with exception at runtime
- 1.2.X: deprecation warnings reported on the nested classifiers
- >=1.3: deprecation warnings elevated to errors

Data class overriding copy

Issue: KT-19618

Components: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, data classes are prohibited to override copy()

Deprecation cycle:

- <1.2: data classes overriding copy() are compiled fine but may fail at runtime/expose strange behavior
- 1.2.X: deprecation warnings reported on data classes overriding copy()
- >=1.3: deprecation warnings elevated to errors

Inner classes inheriting Throwable that capture generic parameters from the outer class

Issue: KT-17981

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, inner classes are not allowed to inherit Throwable

Deprecation cycle:

- <1.2: inner classes inheriting Throwable are compiled fine. If such inner classes happen to capture generic parameters, it could lead to strange code patterns which fail at runtime.
- 1.2.X: deprecation warnings reported on inner classes inheriting Throwable
- >=1.3: deprecation warnings elevated to errors

Visibility rules regarding complex class hierarchies with companion objects

Issues: KT-21515, KT-25333

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, rules of visibility by short names are stricter for complex class hierarchies involving companion objects and nested classifiers.

Deprecation cycle:

- <1.2: old visibility rules (see Issue for details)
- 1.2.X: deprecation warnings reported on short names which are not going to be accessible anymore. Tooling suggests automated migration by adding full name.
- >=1.3: deprecation warnings elevated to errors. Offending code should add full qualifiers or explicit imports

Non-constant vararg annotation parameters

Issue: KT-23153

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, setting non-constant values as vararg annotation parameters is prohibited

Deprecation cycle:

- <1.2: the compiler allows to pass non-constant value for vararg annotation parameter, but actually drops that value during bytecode generation, leading to non-obvious behavior
- 1.2.X: deprecation warnings reported on such code patterns
- >=1.3: deprecation warnings elevated to errors

Local annotation classes

Issue: KT-23277

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 local annotation classes are not supported

Deprecation cycle:

- <1.2: the compiler compiled local annotation classes fine
- 1.2.X: deprecation warnings reported on local annotation classes
- >=1.3: deprecation warnings elevated to errors

Smartcasts on local delegated properties

Issue: KT-22517

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 smartcasts on local delegated properties are not allowed

Deprecation cycle:

- <1.2: the compiler allowed to smartcast local delegated property, which could lead to unsound smartcast in case of ill-behaved delegates
- 1.2.X: smartcasts on local delegated properries are reported as deprecated (the compiler issues warnings)
- >=1.3: deprecation warnings elevated to errors

mod operator convention

Issues: KT-24197

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 declaration of mod operator is prohibited, as well as calls which resolve to such declarations

Deprecation cycle:

- 1.1.X, 1.2.X: report warnings on declarations of operator mod, as well as on calls which resolve to it
- 1.3.X: elevate warnings to error, but still allow to resolve to operator mod declarations
- 1.4.X: do not resolve calls to operator mod anymore

Passing single element to vararg in named form

Issues: KT-20588, KT-20589. See also KT-20171

Component: Core language

Incompatible change type: Source

Short summary: in Kotlin 1.3, assigning single element to vararg is deprecated and should be replaced with consecutive spread and array construction.

Deprecation cycle:

- <1.2: assigning one value element to vararg in named form compiles fine and is treated as assigning single element to array, causing non-obvious behavior when assigning array to vararg
- 1.2.X: deprecation warnings are reported on such assignments, users are suggested to switch to consecutive spread and array construction.
- 1.3.X: warnings are elevated to errors
- >= 1.4: change semantic of assigning single element to vararg, making assignment of array equivalent to the assignment of a spread of an array

Retention of annotations with target EXPRESSION

Issue: KT-13762

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, only SOURCE retention is allowed for annotations with target EXPRESSION

Deprecation cycle:

- <1.2: annotations with target EXPRESSION and retention other than SOURCE are allowed, but silently ignored at use-sites
- 1.2.X: deprecation warnings are reported on declarations of such annotations
- >=1.3: warnings are elevated to errors

Annotations with target PARAMETER shouldn't be applicable to parameter's type

Issue: KT-9580

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, error about wrong annotation target will be properly reported when annotation with target PARAMETER is applied to parameter's type

Deprecation cycle:

- <1.2: aforementioned code patterns are compiled fine; annotations are silently ignored and not present in the bytecode
- 1.2.X: deprecation warnings are reported on such usages
- >=1.3: warnings are elevated to errors

Array.copyOfRange throws an exception when indices are out of bounds instead of enlarging the returned array

Issue: KT-19489

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, ensure that the toIndex argument of Array.copyOfRange, which represents the exclusive end of the range being copied, is not greater than the array size and throw IllegalArgumentException if it is.

Deprecation cycle:

- <1.3: in case toIndex in the invocation of Array.copyOfRange is greater than the array size, the missing elements in range fill be filled with nulls, violating soundness of the Kotlin type system.
- >=1.3: check that toIndex is in the array bounds, and throw exception if it isn't

Progressions of ints and longs with a step of Int.MIN_VALUE and Long.MIN_VALUE are outlawed and won't be allowed to be instantiated

Issue: KT-17176

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, prohibit step value for integer progressions being the minimum negative value of its integer type (Long or Int), so that calling IntProgression.fromClosedRange(0, 1, step = Int.MIN_VALUE) will throw IllegalArgumentException

Deprecation cycle:

- <1.3: it was possible to create an IntProgression with Int.MIN_VALUE step, which yields two values [0, -2147483648], which is non-obvious behavior
- >=1.3: throw IllegalArgumentException if the step is the minimum negative value of its integer type

Check for index overflow in operations on very long sequences

Issue: KT-16097

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, make sure index, count and similar methods do not overflow for long sequences. See the Issue for the full list of affected methods.

Deprecation cycle:

- <1.3: calling such methods on very long sequences could produce negative results due to integer overflow
- >=1.3: detect overflow in such methods and throw exception immediately

Unify split by an empty match regex result across the platforms

Issue: KT-21049

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, unify behavior of split method by empty match regex across all platforms

Deprecation cycle:

- <1.3: behavior of described calls is different when comparing JS, JRE 6, JRE 7 versus JRE 8+
- >=1.3: unify behavior across the platforms

Discontinued deprecated artifacts in the compiler distribution

Issue: KT-23799

Component: other

Incompatible change type: Binary

Short summary: Kotlin 1.3 discontinues the following deprecated binary artifacts:

- kotlin-runtime: use kotlin-stdlib instead
- kotlin-stdlib-jre7/8: use kotlin-stdlib-jdk7/8 instead
- kotlin-jslib in the compiler distribution: use kotlin-stdlib-js instead

Deprecation cycle:

- 1.2.X: the artifacts were marked as deprecated, the compiler reported warning on usage of those artifacts
- >=1.3: the artifacts are discontinued

Annotations in stdlib

Issue: KT-21784

Component: kotlin-stdlib (JVM)

Incompatible change type: Binary

Short summary: Kotlin 1.3 removes annotations from the package org.jetbrains.annotations from stdlib and moves them to the separate artifacts shipped with the compiler: annotations-13.0.jar and mutability-annotations-compat.jar

Deprecation cycle:

- <1.3: annotations were shipped with the stdlib artifact
- >=1.3: annotations ship in separate artifacts

Compatibility modes

When a big team is migrating onto a new version, it may appear in an "inconsistent state" at some point, when some developers have already updated but others haven't. To prevent the former from writing and committing code that others may not be able to compile, we provide the following command line switches (also available in the IDE and Gradle/Maven):

- -language-version X.Y - compatibility mode for Kotlin language version X.Y, reports errors for all language features that came out later.
- -api-version X.Y - compatibility mode for Kotlin API version X.Y, reports errors for all code using newer APIs from the Kotlin Standard Library (including the code generated by the compiler).

Currently, we support the development for at least three previous language and API versions in addition to the latest stable one.

What is cross-platform mobile development?

Nowadays, many companies are facing the challenge of needing to build mobile apps for multiple platforms, specifically for both Android and iOS. This is why cross-platform mobile development solutions have emerged as one of the most popular software development trends.

According to Statista, there were 3.48 million mobile apps available on the Google Play Store and 2.22 million apps on the App Store in the first quarter of 2021, with Android and iOS now accounting for 99% of the worldwide mobile operating system market.

How do you go about creating a mobile app that can reach Android and iOS audiences? In this article, you will find out why more and more mobile engineers are choosing a cross-platform, or multiplatform, mobile development approach.

Cross-platform mobile development: definition and solutions

Multiplatform mobile development is an approach that allows you to build a single mobile application that runs smoothly on several operating systems. In cross-platform apps, some or even all of the source code can be shared. This means that developers can create and deploy mobile assets that work on both Android and iOS without having to recode them for each individual platform.

Different approaches to mobile app development

There are four main ways to create an application for both Android and iOS.

1. Separate native apps for each operating system

When creating native apps, developers build an application for a particular operating system and rely on tools and programming languages designed specifically for one platform: Kotlin or Java for Android, Objective-C or Swift for iOS.

These tools and languages give you access to the features and capabilities of a given OS and allow you to craft responsive apps with intuitive interfaces. But if you want to reach both Android and iOS audiences, you will have to create separate applications, and that takes a lot of time and effort.

2. Progressive web apps (PWAs)

Progressive web apps combine the features of mobile apps with solutions used in web development. Roughly speaking, they offer a mix of a website and a mobile application. Developers build PWAs using web technologies, such as JavaScript, HTML, CSS, and WebAssembly.

Web applications do not require separate bundling or distribution and can be published online. They are accessible via the browser on your computer, smartphone, and tablet, and don't need to be installed via Google Play or the App Store.

The drawback here is that a user cannot utilize all of their device's functionality, for example, contacts, calendars, the phone, and other assets, which results in a limited user experience. In terms of app performance, native apps have the lead.

3. Cross-platform apps

As mentioned earlier, multiplatform apps are designed to run identically on different mobile platforms. Cross-platform frameworks allow you to write shareable and reusable code for the purpose of developing these apps.

This approach has several benefits, such as efficiency with respect to both time and cost. We'll take a closer look at the pros and cons of cross-platform mobile development in a later section.

4. Hybrid apps

When browsing websites and forums, you may notice that some people use the terms "cross-platform mobile development" and "hybrid mobile development" interchangeably. Doing so, however, is not entirely accurate.

When it comes to cross-platform apps, mobile engineers can write code once and then reuse it on different platforms. Hybrid app development, on the other hand, is an approach that combines native and web technologies. It requires you to embed code written in a web development language like HTML, CSS, or JavaScript into a native app. You can do this with the help of frameworks, such as Ionic Capacitor and Apache Cordova, using additional plugins to get access to the native functionalities of platforms.

The only similarity between cross-platform and hybrid development is code shareability. In terms of performance, hybrid applications are not on par with native apps. Because hybrid apps deploy a single code base, some features are specific to a particular OS and don't function well on others.

Native or cross-platform app development: a longstanding debate

The debate around native and cross-platform development remains unresolved in the tech community. Both of these technologies are in constant evolution and come with their own benefits and limitations.

Some experts still prefer native mobile development over multiplatform solutions, identifying the stronger performance and better user experience of native apps as some of the most important benefits.

However, many modern businesses need to reduce the time to market and the cost of per platform development while still aiming to have a presence both on Android and iOS. This is where cross-platform development frameworks like Kotlin Multiplatform Mobile can help, as David Henry and Mel Yahya, a pair of senior software engineers from Netflix, note:

> The high likelihood of unreliable network connectivity led us to lean into mobile solutions for robust client side persistence and offline support. The need for fast product delivery led us to experiment with a multiplatform architecture. Now we're taking this one step further by using Kotlin Multiplatform to write platform agnostic business logic once in Kotlin and compiling to a Kotlin library for Android and a native Universal Framework for iOS.

Is cross-platform mobile development right for you?

Choosing a mobile development approach that is right for you depends on many factors, like business requirements, objectives, and tasks. Like any other solution, cross-platform mobile development has its pros and cons.

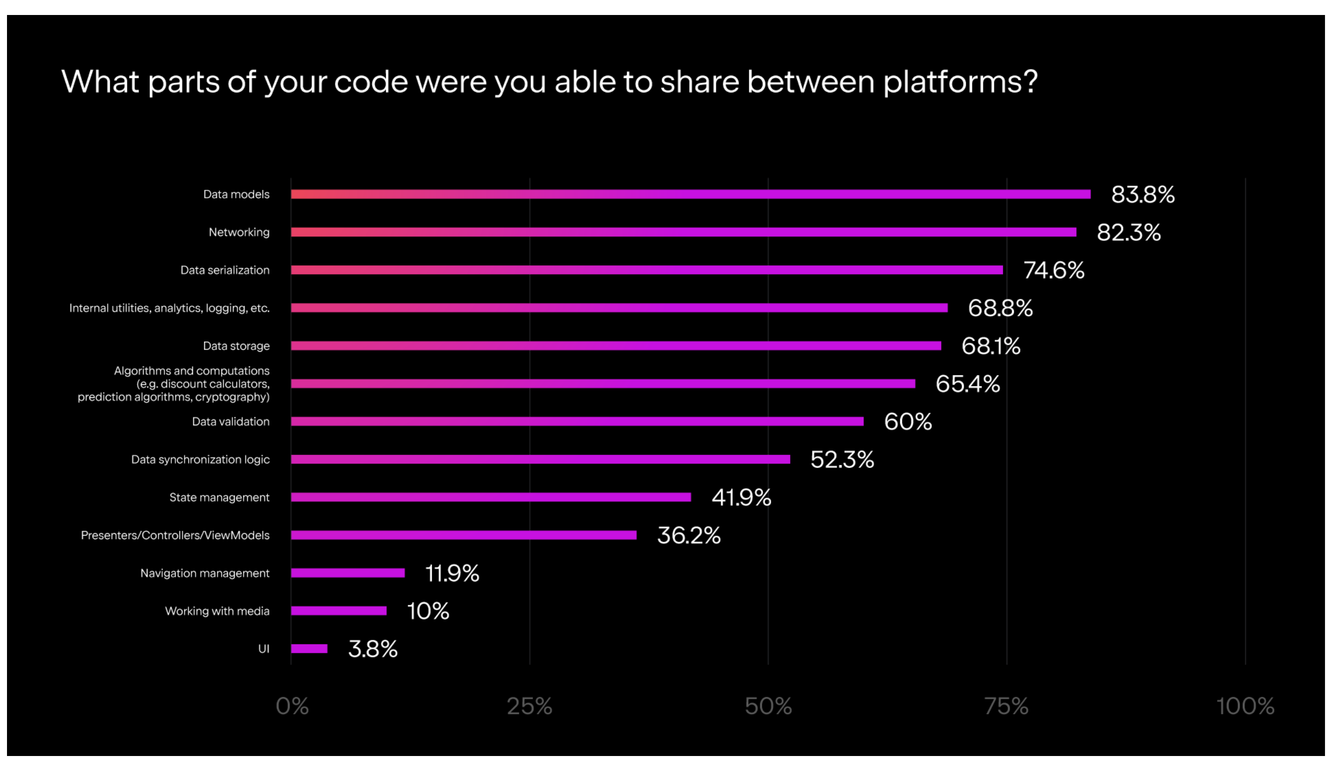
Benefits of cross-platform development

There are plenty of reasons businesses choose this approach over other options.

1. Reusable code

With cross-platform programming, mobile engineers don't need to write new code for every operating system. Using a single codebase allows developers to cut down on time spent doing repetitive tasks, such as API calls, data storage, data serialization, and analytics implementation.

In our Kotlin Multiplatform survey from Q3-Q4 2021, we asked the Kotlin community about the parts of code they were able to share between different platforms.



Parts of code Kotlin Multiplatform Mobile users can share between platforms

2. Time savings

Due to code reusability, cross-platform applications require less code, and when it comes to coding, less code is more. Time saved is because you do not have to write as much code. Additionally, with fewer lines of code, there are fewer places for bugs to emerge, resulting in less time spent testing and maintaining your code.

3. Effective resource management

Building separate applications is expensive. Having a single codebase helps you effectively manage your resources. Both your Android and your iOS development teams can learn how to write and use shared code.

4. Attractive opportunities for developers

Many mobile engineers view modern cross-platform technologies as desirable elements in a product's tech stack. Developers may get bored at work due to

repetitive and routine tasks, such as JSON parsing. However, new technologies and tasks can bring back their excitement, motivation, and joy for work tasks. This means that having a modern tech stack can actually simplify the hiring process for your mobile team.

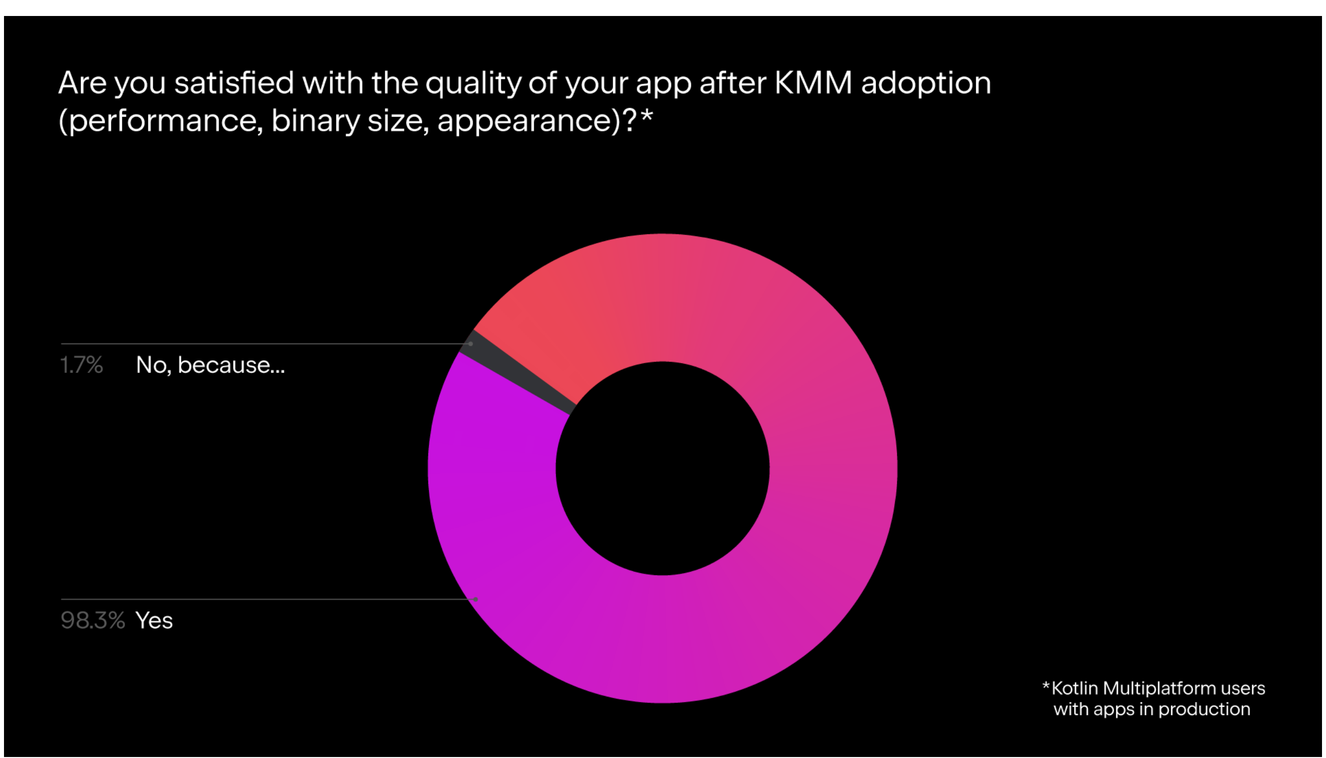
5. Opportunity to reach wider audiences
You don't have to choose between different platforms. Since your app is compatible with multiple operating systems, you can satisfy the needs of both Android and iOS audiences and maximize your reach.

6. Quicker time to market and customization
Since you don't need to build different apps for different platforms, you can develop and launch your product much faster. What's more, if your application needs to be customized or transformed, it will be easier for programmers to make small changes to specific parts of your codebase. This will also allow you to be more responsive to user feedback.

Challenges of a cross-platform development approach

All solutions come with their own limitations. What issues might you encounter with cross-platform programming? Some individuals in the tech community argue that multiplatform development still struggles with glitches related to performance. Furthermore, project leads might have fears that their aim to optimize the development process will have a negative impact on the user experience of an application. However, with improvements to the technologies, cross-platform solutions are becoming increasingly stable, adaptable, and flexible.

In our Kotlin Multiplatform survey from Q1-Q2 2021, we asked survey participants whether they were satisfied with the quality of their apps after adopting Kotlin Multiplatform Mobile. When asked whether they were satisfied with their apps' performance, binary size, and appearance, as many as 98.3% of respondents answered positively.



How are users satisfied with the quality of their app after Kotlin Multiplatform Mobile adoption?

Another concern is the inability to seamlessly support the native features of applications. Nevertheless, if you're building a multiplatform app that needs to access platform-specific APIs, you can use Kotlin's expected and actual declarations. They allow you to define in common code that you "expect" to be able to call the same function across multiple platforms and provide the "actual" implementations, which can interact with any platform-specific libraries thanks to Kotlin interoperability with Java and Objective-C/Swift.

These issues raise the question of whether the end-user will notice a difference between native and cross-platform apps.

As modern multiplatform frameworks continue to evolve, they increasingly allow mobile engineers to craft a native-like experience. If an application is well written, the user will not be able to notice the difference. However, the quality of your product will heavily depend on the cross-platform app development tools you choose.

The most popular cross-platform solutions

The most popular cross-platform frameworks include Flutter, React Native, and Kotlin Multiplatform Mobile. Each of these frameworks has its own capabilities and strengths. Depending on the tool you use, your development process and the outcome may vary.

Flutter

Flutter is a cross-platform development framework that was created by Google and uses the Dart programming language. Flutter supports native features, such as location services, camera functionality, and hard drive access. If you need to create a specific app feature that's not supported in Flutter, you can write platform-specific code using the Platform Channel technology.

Apps built with Flutter need to share all of their UX and UI layers, which is why they may not always feel 100% native. One of the best things about this framework is its Hot Reload feature, which allows developers to make changes and view them instantly.

This framework may be the best option in the following situations:

- You want to share UI components between your apps but you want your applications to look close to native.
- The app is expected to put a heavy load on CPU/GPU.
- You need to develop an MVP application.

Among the most popular apps built with Flutter are Google Ads, Xianyu by Alibaba, eBay Motors, and Hamilton.

React Native

Facebook introduced React Native in 2015 as an open-source framework designed to help mobile engineers build hybrid native/cross-platform apps. It's based on ReactJS – a JavaScript library for building user interfaces. In other words, it uses JavaScript to build mobile apps for Android and iOS systems.

React Native provides access to several third-party UI libraries with ready-to-use components, helping mobile engineers save time during the development process. Like Flutter, it allows you to see all your changes immediately, thanks to the Fast Refresh feature.

You should consider using React Native for your app in the following cases:

- Your application is relatively simple and is expected to be lightweight.
- The development team is fluent in JavaScript or React.

Applications built with React Native include Facebook, Instagram, Skype, and Uber Eats.

Kotlin Multiplatform Mobile

Kotlin Multiplatform Mobile is an SDK for cross-platform mobile development provided by JetBrains. It allows you to create Android and iOS apps with shared logic. Its key benefits include:

- Smooth integration with existing projects.
- Full control over the UI, along with the ability to use the latest UI frameworks, such as SwiftUI and Jetpack Compose.
- Easy access to Android and iOS SDKs without any restrictions.

> Share the logic of your iOS and Android apps. Get started with Kotlin Multiplatform Mobile.

Global companies and start-ups alike have already leveraged Kotlin Multiplatform Mobile to optimize and accelerate their mobile development efforts. The benefits of this approach are apparent from the stories of the companies that have already adopted it.

- The development team from the award-winning to-do list app Todoist started using Kotlin Multiplatform Mobile to synchronize their app's sorting logic on multiple platforms, and in doing so they combined the benefits of creating cross-platform and native apps. You can learn more about their experience in this video.
- The introduction of Kotlin Multiplatform allowed Philips to become faster at implementing new features and increased the interaction between their Android and iOS developers.
- Shopify was able to use Kotlin Multiplatform to share an astounding 95% of their code, which also delivered a significant performance improvement. Similarly, the startup company Down Dog is using Kotlin Multiplatform to increase the development speed for the apps by maximizing the amount of code shared between

all the platforms: JVM, Native, and JS.

Conclusion

As cross-platform development solutions continue to evolve, their limitations have begun to pale in comparison to the benefits they provide. A variety of technologies are available on the market, all suited to different sets of workflows and requirements. Each of the tools discussed in this article offers extensive support for teams thinking about giving cross-platform a try.

Ultimately, carefully considering your specific business needs, objectives, and tasks, and developing clear goals that you want to achieve with your app, will help you identify the best solution for you.

Native and cross-platform app development: how to choose?

People spend much of their waking time on their mobile devices. They also spend 4.8 hours per day on mobile applications, which makes them attractive to all kinds of businesses.

Mobile app development is constantly evolving, with new technologies and frameworks emerging every year. With various solutions on the market, it's often difficult to choose between them. You might have heard about the long-standing "native versus cross-platform" debate.

There are many factors to consider before building an app, such as development cost, time, and app functionality. This is especially true if you want to target both Android and iOS audiences. It may be challenging to decide which mobile development approach will be the best for your particular project. To help you choose between native and cross-platform app development, we've created a list of six essential things to keep in mind.

What is native mobile app development?

Native mobile development means that you build an app for a particular mobile operating system – in most cases Android or iOS. While working on native applications, developers use specific programming languages and tools. For example, you can create a native Android application with Kotlin or Java, or build an app for iOS with Objective-C or Swift.

Here are the core benefits and limitations.

Benefits	Limitations
High performance. The core programming language and APIs used to build native apps make them fast and responsive.	High cost. Native app development requires two separate teams with different sets of skills, which adds to the time and cost of the development process.
Intuitive user experience. Mobile engineers develop native apps using native SDKs, which makes the UI look consistent. The interfaces of native apps are designed to work well with a specific platform, which makes them feel like an integrated part of the device and provides a more intuitive user experience.	Big development team. Managing large teams of multiple specialists can be challenging. The more people you have working on one project, the greater the effort required for communication and collaboration.
Access to the full feature set of a particular device. Native apps built for a particular operating system have direct access to the device's hardware, such as camera, microphone, and GPS location support.	More errors in code. More lines of code can leave more room for bugs.
	Risks of having different logic on Android and iOS apps. With native app development, the code written for one mobile platform cannot be tailored to another platform. For instance, Android and iOS apps might show different prices for the same item because of a mistake in the way the discount is calculated.

What is cross-platform app development?

Cross-platform app development, also called multiplatform development, is the process of building mobile apps that are compatible with several operating systems. Instead of creating separate applications for iOS and Android, mobile engineers can share some or all of the source code between multiple platforms. This way, the applications will work the same on both iOS and Android.

There are various open-source frameworks for cross-platform mobile app development available today. Some of the most popular are Flutter, React Native, and Kotlin Multiplatform Mobile. Below are some of the key pros and cons.

Benefits	Limitations
Shareable code. Developers create a single codebase without the need to write new code for each OS.	Performance issues. Some developers argue that the performance of multiplatform applications is low compared to native apps.
Faster development. You don't need to write or test as much code, which can help you accelerate the development process.	Difficult to access native features of mobile devices. Building a cross-platform app that needs to access platform-specific APIs requires more effort.
Cost-effectiveness. The cross-platform solution can be a good option to consider for startups and companies with smaller budgets because it allows them to reduce development costs.	Limited UI consistency. With cross-platform development frameworks that allow you to share the UI, applications may look and feel less native.
New work opportunities. You can attract new talent to your team with modern cross-platform technologies in a product's tech stack. Many developers want to tackle new challenges at work, which is why new technologies and tasks tend to increase developers' motivation and enjoyment while working.	Challenging hiring process. It can be harder to find professionals who can build multiplatform apps, compared to native app developers. For example, while writing this article, we found about 2,400 Android developer jobs versus 348 Flutter developer vacancies on Glassdoor. However, this situation may change as cross-platform technologies continually evolve and attract more mobile engineers.
Shared logic. Because this approach involves using a single codebase, you can be sure that you have the same application logic on different platforms.	

These are just a few of the key advantages of cross-platform app development. You can learn more about its benefits and use cases from global companies in our article about cross-platform mobile development. As for the challenges of the approach – we'll discuss those in the following section.

Debugging some popular myths about cross-platform app development

Multiplatform technology is constantly evolving. Some cross-platform development frameworks like Kotlin Multiplatform Mobile provide the benefits of building both cross-platform and native apps and remove the limitations that are commonly associated with the approach.

1. Cross-platform apps provide poorer performance than native apps.

Poor performance was long considered to be one of the main disadvantages of multiplatform applications. However, the performance and quality of your product largely depend on the tools you use to build the app. The latest cross-platform frameworks provide all the tools necessary to develop apps with a native-like user experience.

By using different compiler backends, Kotlin is compiled to platform formats – JVM bytecode for Android and native binaries for iOS. As a result, the performance of your shared code is the same as if you write them natively.

2. Cross-platform frameworks are unsafe.

There's a common misconception that native apps are much more secure and reliable. However, modern cross-platform development tools allow developers to build safe apps that guarantee reliable data protection. Mobile engineers just need to take additional measures to make their apps secure.

3. Cross-platform apps don't have access to all native functions of mobile devices.

It is true that not all cross-platform frameworks allow you to build apps with full access to the device's features. Nevertheless, some modern multiplatform frameworks can help you overcome this challenge. For example, Kotlin Multiplatform Mobile gives easy access to Android and iOS SDKs. It provides a Kotlin mechanism of expected and actual declarations to help you access the device's capabilities and features.

4. It can often be difficult to manage cross-platform projects.
In fact, it's the opposite. Cross-platform solutions help you more effectively manage resources. Your development teams can learn how to write and reuse shared code. Android and iOS developers can achieve high efficiency and transparency by interacting and sharing knowledge.

Six key aspects to help you choose between cross-platform app development and the native approach

Now, let's take a look at important factors you need to consider when choosing between native and cross-platform solutions for mobile app development.

1. The type and purpose of your future app
One of the first steps is understanding what app you will be building, including its features and purpose. A complex application with many features will require a lot of programming, especially if it's something new that doesn't have any existing templates.

How crucial is the user interface of your app? Are you looking for outstanding visuals or is the UI less important? Will it require any specific hardware functionality and access to camera and GPS location support? You need to make sure the mobile development approach you choose provides the necessary tools to build the app you need and provide a great user experience.

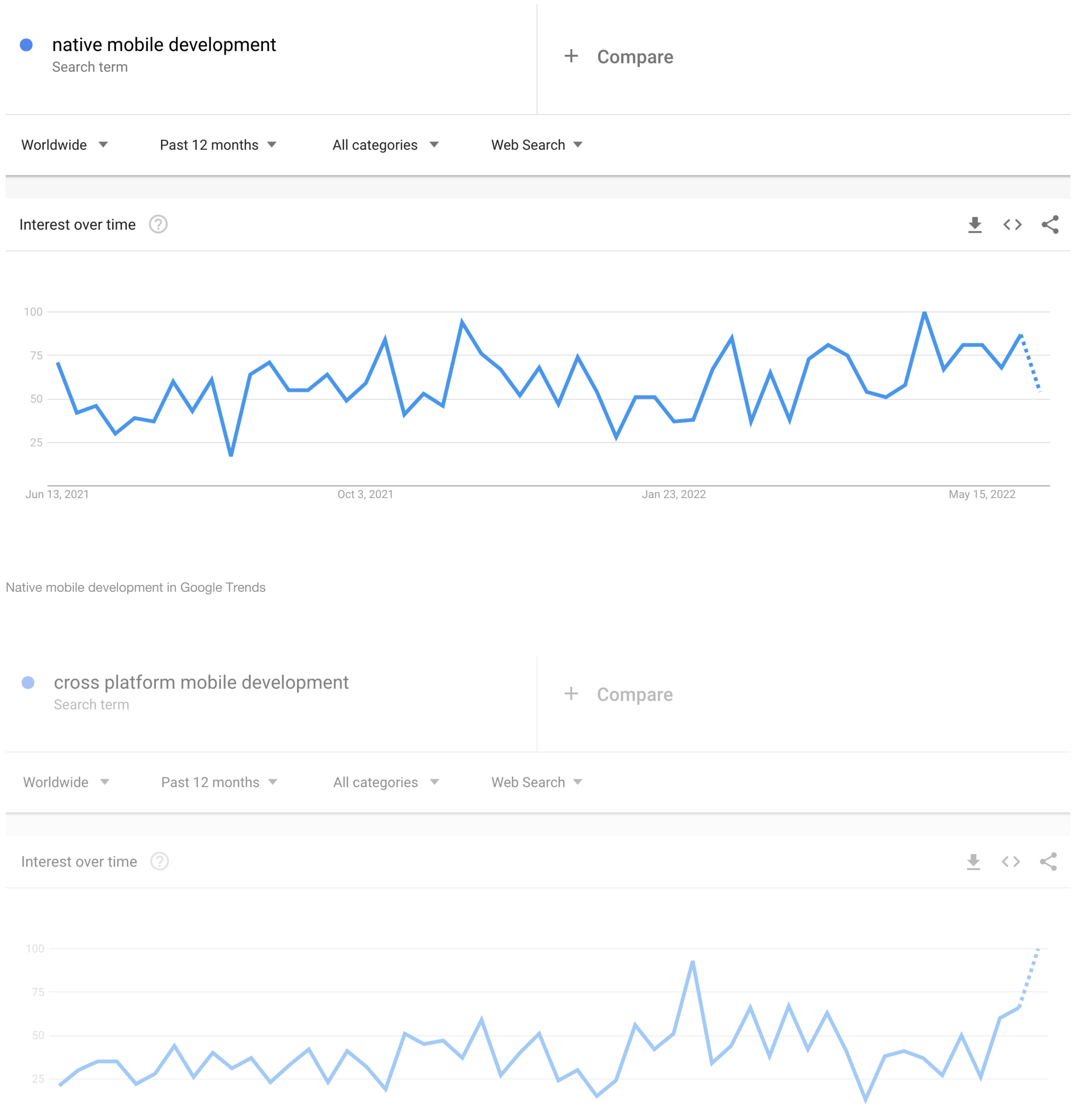
2. Your team's experience in programming languages and tools
The developers on your team should have enough experience and expertise to work with particular frameworks. Pay close attention to what programming skills and languages the development tools require.

For example, developers need to know Objective-C or Swift to create native apps for iOS, and they need to know Kotlin or Java for Android. The cross-platform framework Flutter requires knowledge of Dart. If you use Kotlin Multiplatform Mobile, Kotlin syntax is easy for iOS developers to learn because it follows concepts similar to Swift.

3. Long-term viability
When choosing between different approaches and frameworks, you need to be confident that the platform vendor will continue supporting it over the long term. You can dig into the details about the provider, the size of their community, and adoption by global companies. For example, Kotlin Multiplatform Mobile was developed by JetBrains, Flutter by Google, and React Native by Facebook.

4. Development cost and your budget
As mentioned above, different mobile development solutions and tools come with different expenses. Depending on how flexible your budget is, you can choose the right solution for your project.

5. Adoption in the industry
You can always find out what other experts in the tech community are saying about different approaches. Reddit, StackOverflow, and Google Trends are a few good resources. Just take a look at search trends for the following two terms: "native mobile development" versus "cross-platform mobile development". Many users are still interested in learning about native app development, but it also seems like the cross-platform approach is gaining popularity.



Native mobile development in Google Trends

Cross-platform mobile development in Google Trends

If a technology is widely used by professionals, it has a strong ecosystem, many libraries, and best practices from the tech community, which makes development faster.

6. Visibility and learning resources

If you're considering trying cross-platform app development, one of the factors you should consider is how easy it is to find learning materials for the different multiplatform frameworks. Check their official documentation, books, and courses. Be sure they provide a product roadmap with long-term plans.

When should you choose cross-platform app development?

Cross-platform solutions for mobile app development will save you time and effort when building applications for both Android and iOS.

In a nutshell, you should to opt for cross-platform solutions if:

- You need to build an app for both Android and iOS.
- You want to optimize development time.
- You want to have a single codebase for the app logic while keeping full control over UI elements. Not all cross-platform frameworks allow you to do this, but some, like Kotlin Multiplatform Mobile, provide this capability.
- You're eager to embrace a modern technology that continues to evolve.

> Share the logic of your iOS and Android apps. See Kotlin Multiplatform Mobile in action.

When should you choose native app development?

There may be a few specific cases when it makes sense to choose native mobile development. You should choose this approach if:

- Your app is targeting one specific audience – either Android or iOS.
- The user interface is critical to your future application. However, even if you take the native approach, you can try using multiplatform mobile app development solutions that allow you to share app logic, but not the UI, for your project.
- Your team is equipped with highly skilled Android and iOS developers, but you don't have time to introduce new technologies.

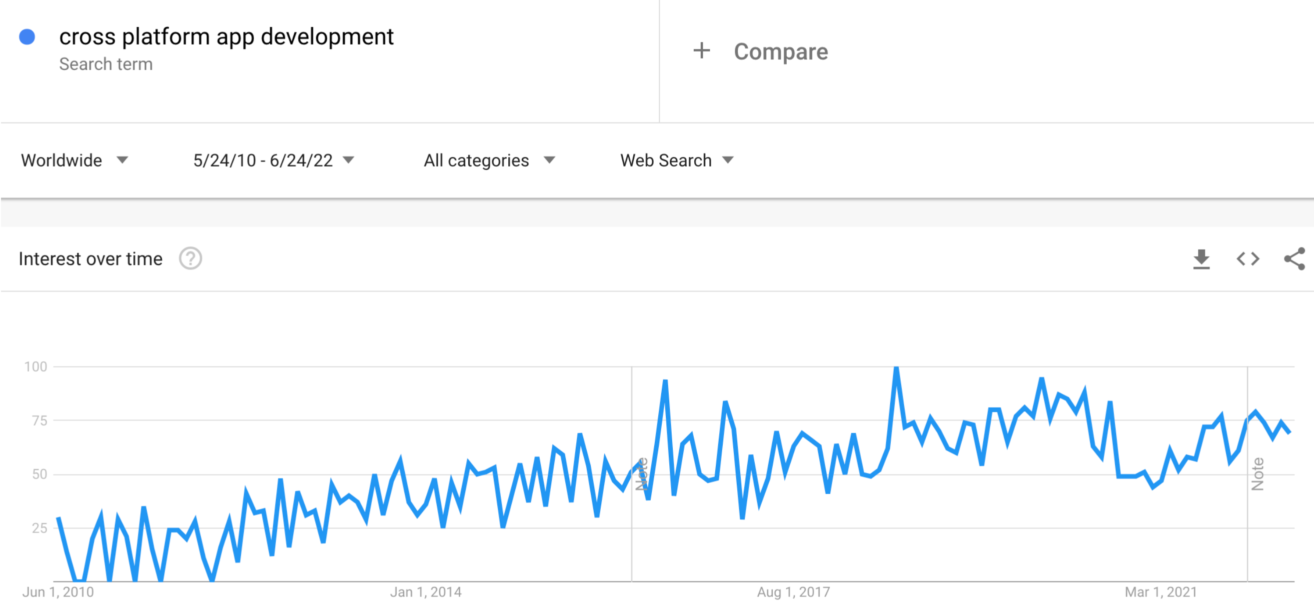
Takeaways
Keep in mind all the aspects described above, your project's goals, and the end user. Whether you're better off with native or cross-platform development depends on your unique needs. Each solution has its strengths and weaknesses.

Nevertheless, keep an eye on what happens in the community. Knowing the latest mobile development trends will help you make the best choice for your project.

The Six Most Popular Cross-Platform App Development Frameworks

Over the years, cross-platform app development has become one of the most popular ways to build mobile applications. A cross-platform, or multiplatform, approach allows developers to create apps that run similarly on different mobile platforms.

Interest has steadily increased over the period from 2010 to date, as this Google Trends chart illustrates:



Google Trends chart illustrating the interest in cross-platform app development

The growing popularity of the rapidly advancing cross-platform mobile development technology has resulted in many new tools emerging on the market. With many options available, it can be challenging to pick the one that will best suit your needs. To help you find the right tool, we've put together a list of the six best cross-platform app development frameworks and the features that make them great. At the end of this article, you will also find a few key things to pay attention to when choosing a multiplatform development framework for your business.

What is a cross-platform app development framework?

Mobile engineers use cross-platform mobile development frameworks to build native-looking applications for multiple platforms, such as Android and iOS, using a single codebase. Shareable code is one of the key advantages this approach has over native app development. Having one single codebase means that mobile engineers can save time by avoiding the need to write code for each operating system, accelerating the development process.

With demand for cross-platform solutions for mobile app development growing, the number of tools available on the market is increasing as well. In the following section, we provide an overview of the most widely used frameworks for building cross-platform mobile apps for iOS, Android, and other platforms. Our summaries include the programming languages these frameworks are based on, as well as their main features and advantages.

Popular cross-platform app development frameworks

This list of tools is not exhaustive; many other options are available on the market today. The important thing to realize is that there's no one-size-fits-all tool that will be ideal for everyone. The choice of framework largely depends on your particular project and your goals, as well as other specifics that we will cover at the end of the article.

Nevertheless, we've tried to pick out some of the best frameworks for cross-platform mobile development to give you a starting point for your decision.

Flutter
Released by Google in 2017, Flutter is a popular framework for building mobile, web, and desktop apps from a single codebase. To build applications with Flutter, you will need to use Google's programming language called Dart.

Programming language: Dart.

Mobile apps: eBay, Alibaba, Google Pay, ByteDance apps.

Key features:

- Flutter's hot reload feature allows you to see how your application changes as soon as you modify your code, without you having to recompile it.
- Flutter supports Google's Material Design, a design system that helps developers build digital experiences. You can use multiple visual and behavioral widgets

when building your app.

- Flutter doesn't rely on web browser technology. Instead, it has its own rendering engine for drawing widgets.

Flutter has a relatively active community of users around the world. It is widely used by many developers. According to the Stack Overflow Developer Survey 2021, Flutter is the second most-loved framework.

React Native

An open-source UI software framework, React Native was developed in 2015 (a bit earlier than Flutter) by Meta Platforms, formerly Facebook. It's based on Facebook's JavaScript library React and allows developers to build natively rendered cross-platform mobile apps.

Programming language: JavaScript.

Mobile apps: Skype, Bloomberg, Shopify, various small modules in Facebook and Instagram.

Key features:

- Developers can see their changes in their React components immediately, thanks to the Fast Refresh feature.
- One of React Native's advantages is a focus on the UI. React primitives render to native platform UI components, allowing you to build a customized and responsive user interface.
- In versions 0.62 and higher, integration between React Native and the mobile app debugger Flipper is enabled by default. Flipper is used to debug Android, iOS, and React native apps, and it provides tools like a log viewer, an interactive layout inspector, and a network inspector.

As one of the most popular cross-platform app development frameworks, React Native has a large and strong community of developers who share their technical knowledge. Thanks to this community, you can get the support you need when building mobile apps with the framework.

Kotlin Multiplatform Mobile

Kotlin Multiplatform Mobile is an SDK developed by JetBrains for creating Android and iOS applications. It allows you to share common code between the two platforms and write platform-specific code only when it's necessary, for example, when you need to build native UI components or when you are working with platform-specific APIs.

Programming language: Kotlin.

Mobile apps: Philips, Baidu, Netflix, Leroy Merlin.

Key features:

- You can easily start using Kotlin Multiplatform Mobile in existing projects.
- Kotlin Multiplatform Mobile provides you with full access over the user interface. You can utilize the latest UI frameworks, such as SwiftUI and Jetpack Compose.
- Developers have easy access to the Android and iOS SDKs without any restrictions.

Even though this cross-platform mobile development framework is the youngest on our list, it has a mature community. It's growing fast and is already making a distinct impression on today's market. Thanks to its regularly updated documentation and community support, you can always find answers to your questions. What's more, many global companies and startups already use Kotlin Multiplatform Mobile to develop multiplatform apps with a native-like user experience.

> Create your first cross-platform mobile app with Kotlin Multiplatform Mobile.

Ionic

Ionic is an open-source UI toolkit that was released in 2013. It helps developers build hybrid mobile and desktop applications using a combination of native and web technologies, like HTML, CSS, and JavaScript, with integrations for the Angular, React, and Vue frameworks.

Programming language: JavaScript.

Mobile apps: T-Mobile, BBC (Children's & Education apps), EA Games.

Key features:

- Ionic is based on a SaaS UI framework designed specifically for mobile OS and provides multiple UI components for building applications.
- The Ionic framework uses the Cordova and Capacitor plugins to provide access to device's built-in features, such as the camera, flashlight, GPS, and audio

recorder.

- Ionic has its own IDE called Ionic Studio, which was designed for building and prototyping apps with minimal coding.

There's constant activity on the Ionic Forum, where community members exchange knowledge and help each other overcome their development challenges.

Xamarin

Xamarin was launched in 2011 and is now owned by Microsoft. It's an open-source cross-platform app development framework that uses the C# language and the .Net framework to develop apps for Android, iOS, and Windows.

Programming language: C#.

Mobile apps: UPS, Alaska Airlines, Academy Members (Academy of Motion Picture Arts and Sciences).

Key features:

- Xamarin applications use the Base Class Library, or .NET BCL, a large collection of classes that have a range of comprehensive features, including XML, database, IO, and networking support, and more. Existing C# code can be compiled for use in your app, giving you access to many libraries that add functionality beyond the BCL.
- With Xamarin.Forms, developers can utilize platform-specific UI elements to achieve a consistent look for their apps across different operating systems.
- Compiled bindings in Xamarin.Forms improve data binding performance. Using these bindings provides compile-time validation for all binding expressions. Because of this feature, mobile engineers get fewer runtime errors.

Xamarin is supported by many contributors across the globe and is especially popular among C, C++, and C# developers who create mobile applications.

NativeScript

This open-source mobile application development framework was initially released in 2014. NativeScript allows you to build Android and iOS mobile apps using JavaScript or languages that transpile to JavaScript, like TypeScript, and frameworks like Angular and Vue.js.

Programming language: JavaScript, TypeScript.

Mobile apps: Daily Nanny, Strudel, Breethe.

Key features:

- NativeScript allows developers to easily access native Android and iOS APIs.
- The framework renders platform-native UIs. Apps built with NativeScript run directly on a native device without relying on WebViews, a system component for the Android OS that allows Android applications to show content from the web inside an app.
- NativeScript offers various plugins and pre-built app templates, eliminating the need for third-party solutions.

NativeScript is based on well-known web technologies like JavaScript and Angular, which is why many developers choose this framework. Nevertheless, it's usually used by small companies and startups.

How do you choose the right cross-platform app development framework for your project?

There are other cross-platform frameworks besides those mentioned above, and new tools will continue to appear on the market. Given the wide array of options, how can you find the right one for your next project? The first step is to understand your project's requirements and goals, and to get a clear idea of what you want your future app to look like. Next, you'll want to take the following important factors into account so you can decide on the best fit for your business.

1. The expertise of your team

Different cross-platform mobile development frameworks are based on different programming languages. Before adopting a framework, check what skills it requires and make sure your team of mobile engineers has enough knowledge and experience to work with it.

For example, if your team is equipped with highly skilled JavaScript developers, and you don't have enough resources to introduce new technologies, it may be worth choosing frameworks that use this language, such as React Native.

2. Vendor reliability and support

It's important to be sure that the maintainer of the framework will continue to support it in the long run. Learn more about the companies that develop and support the frameworks you're considering, and take a look at the mobile apps that have been built using them.

3. UI customization
Depending on how crucial the user interface is for your future app, you may need to know how easily you can customize the UI using a particular framework. For example, Kotlin Multiplatform Mobile provides you with full control over the UI and the ability to use the latest UI frameworks, such as SwiftUI and Jetpack Compose.

4. Framework maturity
Find out how frequently the public API and tooling for a prospective framework changes. For example, some changes to native operating system components break internal cross-platform behavior. It's better to be aware of possible challenges you may face when working with the mobile app development framework. You can also browse GitHub and check how many bugs the framework has and how these bugs are being handled.

5. Framework capabilities
Each framework has its own capabilities and limitations. Knowing what features and tools a framework provides is crucial to identifying the best solution for you. Does it have code analyzers and unit testing frameworks? How quickly and easily will you be able to build, debug, and test your app?

6. Consistency between different platforms
Providing consistency between multiple platforms can be challenging, given how much platforms like Android and iOS significantly differ, particularly in terms of the development experience. For example, tools and libraries aren't the same on these operating systems, so there may be many differences when it comes to the business logic. Some technologies, like Kotlin Multiplatform Mobile, allow you to write and share the app's business logic between Android and iOS platforms.

7. Security
Security and privacy are especially important when building a critical mobile app for business, for example, banking and e-commerce apps that include a payment system. According to OWASP Mobile Top 10, among the most critical security risks for mobile applications are insecure data storage, authentication, and authorization.

You need to ensure that the multiplatform mobile development framework of your choice provides the required level of security. One way to do this is to browse the security tickets on the framework's issue tracker if it has one that's publicly available.

8. Educational materials
The volume and quality of available learning resources about a framework can also help you understand how smooth your experience will be when working with it. Comprehensive official documentation, online and offline conferences, and educational courses are a good sign that you will be able to find enough essential information about a product when you need it.

Key takeaways

Without considering these factors, it's difficult to choose the framework for cross-platform mobile development that will best meet your specific needs. Take a closer look at your future application requirements and weigh them against capabilities of various frameworks. Doing so will allow you to find the right cross-platform solution to help you deliver high-quality apps.

Google Summer of Code with Kotlin

This article contains the list of project ideas for Google Summer of Code with Kotlin, and contributor guidelines.

> Kotlin resources:
>
> - Kotlin GitHub repository
> - Kotlin Slack and the #gsoc Slack channel
>
> If you got any questions, contact us via gsoc@kotlinfoundation.org

Kotlin contributor guidelines for Google Summer of Code (GSoC)

Getting started

1. Check out the GSoC FAQ and the program announcement.
2. Familiarize yourself with the Kotlin language:
   - The official Kotlin website is a great place to start.
   - Read the official documentation to get a better understanding of the language.
   - Take a look at the Kotlin courses on JetBrains Academy or the Android team's Training options.
   - Follow the Kotlin Twitter account to stay up to date on the latest news and developments.
   - Check out the Kotlin YouTube channel for tutorials, tips, and the latest updates.
3. Get to know the Kotlin open source community:
   - Explore the general Kotlin contribution guidelines.
   - Join the Kotlin Slack channel to connect with other developers and get help with any questions you may have.
   - Join the #gsoc channel to ask questions and get support from the GSoC team.

How to apply

> Applications are accepted from March 20 to April 4, 2023.

1. Check out the project ideas and select the one you would like to work on.
2. If you are not familiar with Kotlin, read the introductory info on the Kotlin website.
3. Refer to the GSoC contributor guidelines.
4. Apply via the GSoC website.
   - We suggest that you write a working code sample relevant to the proposed project. You can also show us any code sample that you are particularly proud of.
   - Describe why you are interested in Kotlin and your experience with it.
   - If you participate in open source projects, please reference your contribution history.
   - If you have a GitHub, Twitter account, blog, or portfolio of technical or scientific publications, please reference them as well.
   - Disclose any conflicts with the GSoC timeline due to other commitments, such as exams and vacations.

Thank you! We look forward to reading your applications!

Project ideas

Kotlin Multiplatform protobufs [Hard, 350 hrs]

Description

Add support for Kotlin/Common protos to protoc with Kotlin/Native (iOS) runtime and Objective-C interop.

Motivation

While protobufs have many platform implementations, there isn't a way to use them in Kotlin Multiplatform projects.

Expected outcomes

Design and build Kotlin Multiplatform Protobuf support, culminating in contributions to:

* GitHub – protocolbuffers/protobuf: Protocol Buffers – Google's data interchange format
* GitHub – google/protobuf-gradle-plugin
* Kotlin Multiplatform Gradle Plugin

Skills required (preferred)

* Kotlin
* Objective-C
* C++

Kotlin Compiler error messages [Hard, 350 hrs]

Description

Add improved compiler error messages to the K2 Kotlin compiler: more actionable and detailed information (like Rust has).

Motivation

Rust compiler error messages are often regarded as being by far the most helpful of any compiler. The Kotlin K2 compiler provides a great foundation for better compiler errors in Kotlin but this potential is somewhat untapped.

Expected outcomes

Using StackOverflow and other data sources, uncover common compiler errors which would have significant value to users. Make contributions back to the compiler to improve those error messages.

Skills required (preferred)

* Kotlin
* Compiler architecture

Kotlin Multiplatform libraries [Easy or Medium, 175 or 350 hrs]

Description

Create and deliver (to Maven Central) Kotlin Multiplatform libraries that are commonly needed. For instance, compression, crypto.

Motivation

Kotlin Multiplatform is still fairly new and could use some additional libraries which are either platform independent (Kotlin/Common) and/or have platform implementations (expect/actual).

Expected outcomes

Design and deliver at least one Kotlin Multiplatform library with a greater priority on JVM/Android and Kotlin/Native (iOS) than other targets (Kotlin/JS).

Skills required (preferred)

* Kotlin
* Objective-C

Groovy to Kotlin Gradle DSL Converter [Medium, 350 hrs]

Description

The project aims to create a Groovy-to-Kotlin converter with a primary focus on Gradle scripts. We will start from basic use cases, such as when a user wants to paste Groovy-style dependency declarations to a Kotlin script and the IDE automatically converts them. Later, we will start supporting more complex code constructs and conversions of complete files.

Motivation

The Kotlin Gradle DSL is gaining popularity, so much so that it will soon become the default choice for building projects with Gradle. However, many documents and resources about Gradle still refer to Groovy, and pasting Groovy samples into build.gradle.kts requires manual editing. Furthermore, many new features around Gradle will be in Kotlin first, and consequently users will migrate from the Groovy DSL to the Kotlin DSL. The automatic code conversion of a build setup will

therefore greatly ease this migration, saving a lot of time.

Expected outcomes

A plugin for IntelliJ IDEA that can convert Groovy code to Kotlin with the main focus on the Gradle DSL.

Skills required (preferred)

- Basic knowledge of Gradle
- Basic knowledge of parsers and how compilers work in general
- Basic knowledge of Kotlin

Eclipse Gradle KTS editing [Medium, 350 hrs]

Description

Improve the experience of editing Gradle Kotlin Scripts (KTS) in Eclipse.

Motivation

IntelliJ IDEA and Android Studio have great support for editing KTS Gradle build scripts, but the Eclipse support is lacking. Ctrl-Click to definition, Code completion, Code error highlighting could all be improved.

Expected outcomes

Make contributions to the Gradle Eclipse plugin that improve the developer experience for editing KTS.

Skills required (preferred)

- Kotlin
- Gradle
- Eclipse platform and plugins

Improve support for parameter forwarding in the Kotlin Plugin for IntelliJ IDEA [Medium, 350 hrs]

Description and motivation

The Kotlin plugin provides Kotlin language support in IntelliJ IDEA and Android Studio. In the scope of this project, you will improve parameter forwarding support for the plugin.

To prefer composition over inheritance is a widely known principle. IntelliJ IDEA provides great support for writing code that uses inheritance (completion and quick-fixes the IDE suggests), but the support for code that uses composition instead of inheritance has yet to be implemented.

The main problem of working with code that heavily uses composition is parameter forwarding. In particular:

- The IDE doesn't suggest completing parameter declarations that can be forwarded as arguments to other functions that currently use default arguments.
- The IDE doesn't rename the chain of forwarded parameters.
- The IDE doesn't provide any quick-fixes that fill in all the required arguments with parameters that can be forwarded.

One notable example where such support would be greatly appreciated is Jetpack Compose. Android's modern tool kit for building UI, Jetpack Compose heavily uses function composition and parameter forwarding. It quickly becomes tedious to work with @Composable functions because they have a lot of parameters. For example, androidx.compose.material.TextField has 19 parameters.

Expected outcomes

- Improved parameter and argument completion suggestions in IntelliJ IDEA.
- Implemented IDE quick-fixes that suggest filling in all the required arguments with parameters with the same names and types.
- The Rename refactoring renames the chain of forwarded parameters.
- All other IDE improvements around parameter forwarding and functions that have a lot of parameters.

Skills required (preferred)

* Knowledge of Kotlin and Java
* Ability to navigate in a large codebase

Enhance the kotlinx-benchmark library API and user experience [Easy, 175 hrs]
Description

kotlinx-benchmark is an open-source library for benchmarking multiplatform code written in Kotlin. It has a barebones skeleton but lacks quality-of-life features, such as fine-grained benchmark configuration (like time units, modes), feature parity between JVM and Kotlin/Native benchmarking, a command-line API, and modern Gradle support. Its documentation, integration tests, and examples are also lagging.

Motivation

The library has already been implemented, but it is sometimes difficult to use correctly and confuses some users. Improving the library's user experience would greatly help the Kotlin community.

Expected outcomes

* The library has clear documentation with usage examples.
* The library API is simple and easy to use.
* Options for benchmarking Kotlin/JVM code are also available for benchmarking code on other platforms.

Skills required (preferred)

* Kotlin
* Gradle internals

Parallel stacks for Kotlin Coroutines in the debugger [Hard, 350 hrs]
Description

Implement Parallel Stacks view for Kotlin coroutines to improve the coroutine debugging experience.

Motivation

Currently, support for coroutines debugging is very limited in IntelliJ IDEA. The Kotlin debugger has the Coroutines Panel that allows a user to view all of the coroutines and their states, but it's not very helpful when debugging an application with lots of coroutines in it. The JetBrains Rider has the Parallel Stacks feature that allows a user to inspect threads and their stack traces in a graph view, which could be a great way of inspecting coroutines.

Expected outcomes

Using the Kotlin coroutines debugger API, develop the IntelliJ IDEA plugin which would add the parallel stacks view for coroutines to the debugger. Find ways to improve the graph representation of coroutines.

Skills required (preferred)

* Kotlin
* Kotlin coroutines
* IntelliJ IDEA plugin development

Security

We do our best to make sure our products are free of security vulnerabilities. To reduce the risk of introducing a vulnerability, you can follow these best practices:

* Always use the latest Kotlin release. For security purposes, we sign our releases published on Maven Central with these PGP keys:
  * Key ID: kt-a@jetbrains.com
  * Fingerprint: 2FBA 29D0 8D2E 25EE 84C1 32C3 0729 A0AF F899 9A87
  * Key size: RSA 3072

- Use the latest versions of your application's dependencies. If you need to use a specific version of a dependency, periodically check if any new security vulnerabilities have been discovered. You can follow the guidelines from GitHub or browse known vulnerabilities in the CVE base.

We are very eager and grateful to hear about any security issues you find. To report vulnerabilities that you discover in Kotlin, please post a message directly to our issue tracker or send us an email.

For more information on how our responsible disclosure process works, please check the JetBrains Coordinated Disclosure Policy.

Kotlin documentation as PDF

Here you can download a PDF version of Kotlin documentation that includes everything except tutorials and API reference.

Download Kotlin 1.8.20 documentation (PDF)

View the latest Kotlin documentation (online)

Contribution

Kotlin is an open-source project under the Apache 2.0 License. The source code, tooling, documentation, and even this web site are maintained on GitHub. While Kotlin is mostly developed by JetBrains, there are hundreds of external contributors to the Kotlin project and we are always on the lookout for more people to help us.

Participate in Early Access Preview

You can help us improve Kotlin by participating in Kotlin Early Access Preview (EAP) and providing us with your valuable feedback.

For every release, Kotlin ships a few preview builds where you can try out the latest features before they go to production. You can report any bugs you find to our issue tracker YouTrack and we will try to fix them before a final release. This way, you can get bug fixes earlier than the standard Kotlin release cycle.

Contribute to the compiler and standard library

If you want to contribute to the Kotlin compiler and standard library, go to JetBrains/Kotlin GitHub, check out the latest Kotlin version, and follow the instructions on how to contribute.

You can help us by completing open tasks. Please keep an open line of communication with us because we may have questions and comments on your changes. Otherwise, we won't be able to incorporate your contributions.

Contribute to the Kotlin IDE plugin

Kotlin IDE plugin is a part of the IntelliJ IDEA repository.

To contribute to the Kotlin IDE plugin, clone the IntelliJ IDEA repository and follow the instructions on how to contribute.

Contribute to other Kotlin libraries and tools

Besides the standard library that provides core capabilities, Kotlin has a number of additional (kotlinx) libraries that extend its functionality. Each kotlinx library is developed in a separate repository, has its own versioning and release cycle.

If you want to contribute to a kotlinx library (such as kotlinx.coroutines or kotlinx.serialization) and tools, go to Kotlin GitHub, choose the repository you are interested in and clone it.

Follow the contribution process described for each library and tool, such as kotlinx.serialization, ktor and others.

If you have a library that could be useful to other Kotlin developers, let us know via feedback@kotlinlang.org.

Contribute to the documentation

If you've found an issue in the Kotlin documentation, feel free to check out the documentation source code on GitHub and send us a pull request. Follow these guidelines on style and formatting.

Please keep an open line of communication with us because we may have questions and comments on your changes. Otherwise, we won't be able to incorporate your contributions.

Create tutorials or videos

If you've created tutorials or videos for Kotlin, please share them with us via feedback@kotlinlang.org.

Translate documentation to other languages

You are welcome to translate the Kotlin documentation into your own language and publish the translation on your website. However, we won't be able to host your translation in the main repository and publish it on kotlinlang.org.

This site is the official documentation for the language, and we ensure that all the information here is correct and up to date. Unfortunately, we won't be able to review documentation in other languages.

Hold events and presentations

If you've given or just plan to give presentations or hold events on Kotlin, please fill out the form. We'll feature them on the event list.

KUG guidelines

A Kotlin User Group, or KUG, is a community that is dedicated to Kotlin and that offers you a place to share your Kotlin programming experience with like-minded people.

To become an KUG, your community should have some specific features shared by every KUG. It should:

- Provide Kotlin-related content, with regular meetups as the main form of activity.
- Host regular events (at least once every 3 months) with open registration and without any restriction for attendance.
- Be driven and organized by the community, and it should not use events to earn money or gain any other business benefits from members and attendees.
- Follow and ensure a code of conduct in order to provide a welcoming environment for attendees of any background and experience (check-out our recommended Code of Conduct).

There are no limits regarding the format for KUG meetups. They can take place in whatever fashion works best for the community, whether that includes presentations, hands-on labs, lectures, hackathons, or informal beer-driven get-togethers.

> For Kotlin User Group brand assets, see Kotlin brand assets documentation.

How to run a KUG?

- In order to promote group cohesion and prevent miscommunication, we recommend keeping to a limit of one KUG per city. Check out the list of KUGs to see if there is already a KUG in your area.
- Use the official KUG logo and branding. Check out the branding guidelines.
- Keep your user group active. Run meetups regularly, at least once every 3 months.
- Announce your KUG meetups at least 2 weeks in advance. The announcement should contain a list of talks and the names of the speakers, as well as the location, timing, and any other crucial info about the event.
- KUG events should be free or, if you need to cover organizing expenses, limit prices to a maximum of 10 USD.
- Your group should have a code of conduct available for all members.

If your community has all the necessary features and follows these guidelines, you are ready to Apply to be a new KUG.

Have a question? Contact us

Support for KUGs from JetBrains

Active KUGs that host at least 1 meetup every 3 months can apply for the community support program, which includes:

- Official KUG branding.
- A special entry on the Kotlin website.
- Free licenses for JetBrains products to raffle off at meetups.
- Priority support for Kotlin events and campaigns.
- Help with recruiting Kotlin speakers for your events.

Support from JetBrains for other tech communities

If you organize any other tech communities, you can apply for support as well. By doing so, you may receive:

- Free licenses for JetBrains products to raffle off at meetups.
- Information about Kotlin official events and campaigns.
- Kotlin stickers.
- Help with recruiting Kotlin speakers for your events.

Kotlin Night guidelines

Kotlin Night is a meetup that includes 3-4 talks on Kotlin or related technologies.

> For Kotlin Night brand assets, see Kotlin brand assets documentation.

Event guidelines

- Please use the branding materials we've provided. Having all events and materials in the same style will help keep the Kotlin Night experience consistent.
- Kotlin Night should be a free event. A minimal fee can be charged to cover expenses, but it should remain a non-profit event.
- The event should be announced publicly and open for all people to attend without any kind of discrimination.
- If you publish the contents of the talks online after the event, they must be free and accessible to everyone, without any sign-up or registration procedures.
- Recordings are optional but recommended, and they should also be made available. If you decide to record the talks, we suggest having a plan to ensure the quality is good.
- The talks should primarily be about Kotlin and should not focus on marketing or sales.
- The event can serve food and drinks optionally.

Event requirements

JetBrains is excited to support your Kotlin Night event. Because we want all events to provide the same high-quality experience, we need organizers to ensure that some basic requirements are met for the event to receive JetBrains support. As an organizer, you are responsible for the following aspects of the event:

1. The location and everything required to host the event, including booking a comfortable venue. Please make sure that:
   - All the participants are aware of the exact date, place, and starting time of the event, along with the event schedule and program.
   - There is enough space as well as food and beverages, if you provide them, for everyone.
   - You have a plan with your speakers. This includes a schedule, topics, abstracts for the talks, and any necessary equipment for the presentations.

2. Content and speakers

   - Feel free to invite presenters from your local community, from neighboring countries, or even from all over the globe. You don't have to have any JetBrains representatives or speakers at your event. However, we are always happy to hear about more Kotlin Nights, so feel free to notify us.

3. Announcements and promotion

   - Announce your event at least three weeks before the date of a meetup.
   - Include the schedule, topics, abstracts, and speaker bios in the announcement.
   - Spread the word on social media.

4. Providing event material to JetBrains after the event

   - We would be glad to announce your event at kotlinlang.org, and we would appreciate it if you provided slides and video materials for a follow-up posting.

JetBrains support

JetBrains provides support with:

- Access to Kotlin Night Branding, which includes the name and logos
- Merchandise, such as stickers and t-shirts for speakers and small souvenirs for attendees
- A listing for the event on the Kotlin Talks page
- Help to reach out to speakers to take part in the event, if necessary
- Help to find a location if possible (via contacts, etc.), as well as help to identify possible partnerships with local businesses

Kotlin brand assets

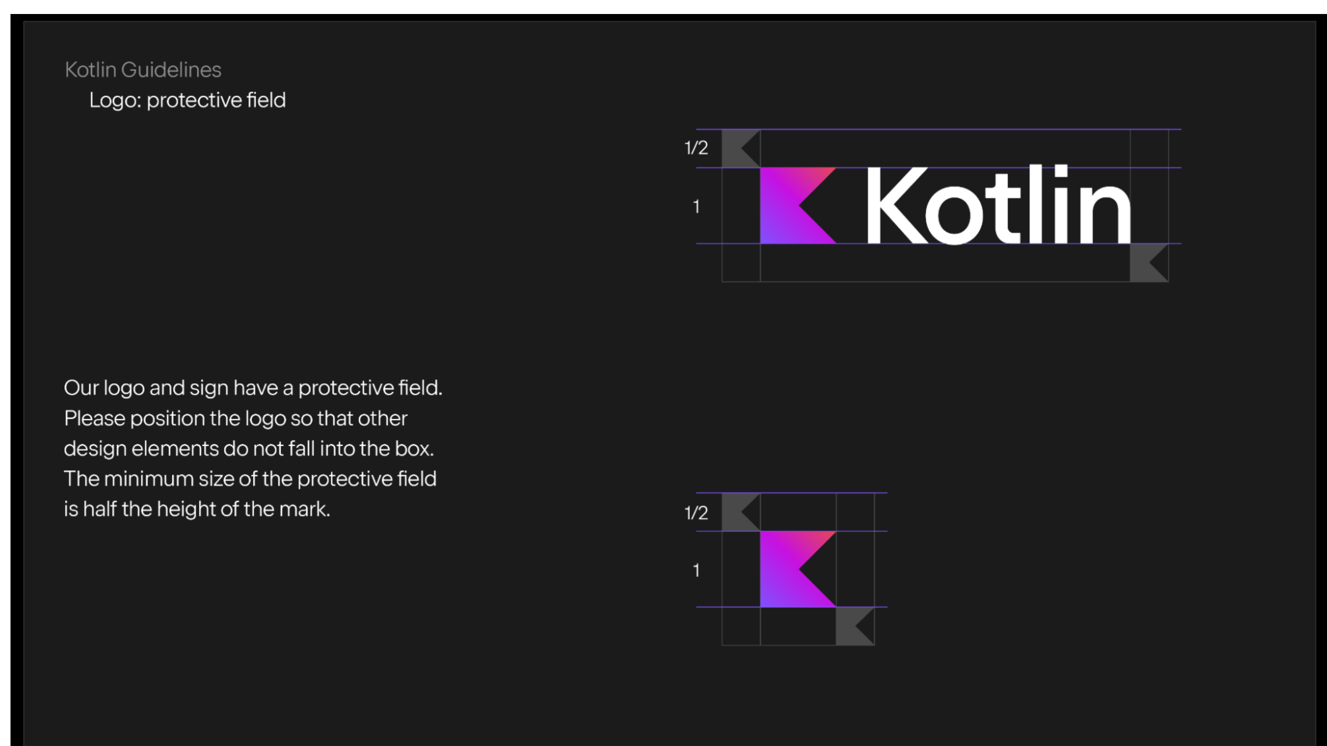
Kotlin Logo

Our logo consists of a mark and a typeface. The full-color version is the main one and should be used in the vast majority of cases.

Download all versions

Kotlin logo

Our logo and mark have a protective field. Please position the logo so that other design elements do not come into the box. The minimum size of the protective field is half the height of the mark.



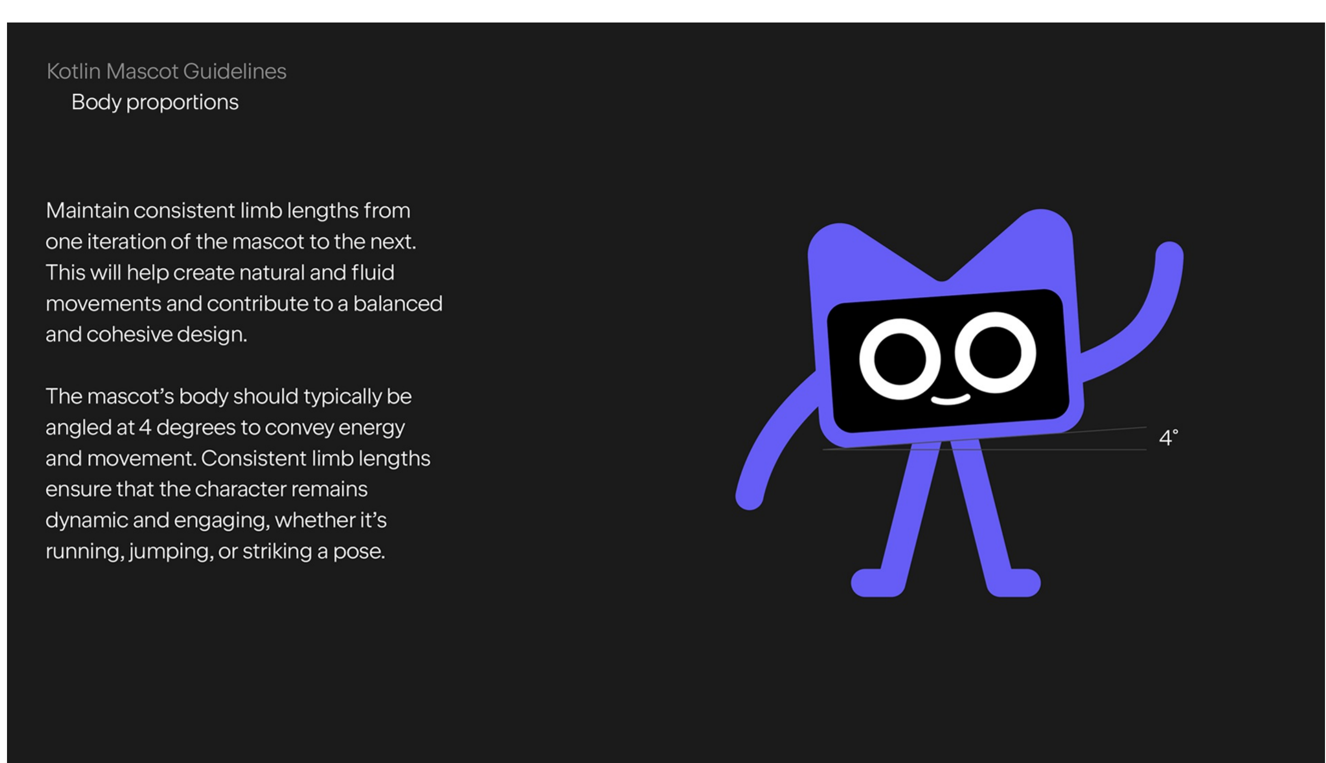
Kotlin logo proportions

Pay special attention to the following restrictions concerning the use of the logo:

- Do not separate the mark from the text. Do not swap elements.
- Do not change the transparency of the logo.
- Do not outline the logo.
- Do not repaint the logo in third-party colors.
- Do not change the text.
- Do not set the logo against a complex background. Do not place the logo in front of a bright background.

Kotlin mascot

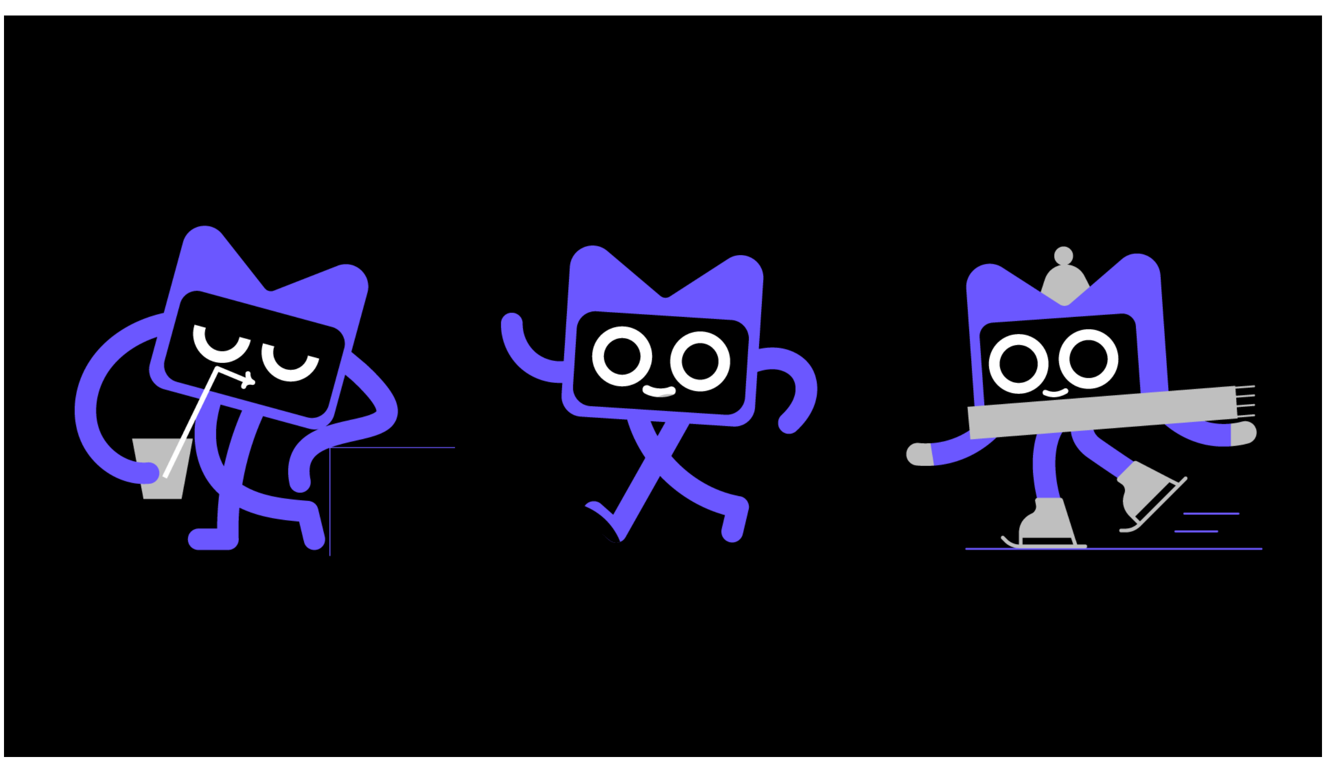
Kodee is Kotlin's reimagined mascot. More than just a symbol, Kodee is your friendly companion who's there to encourage and inspire you to express your creativity. When using it, we ask you to follow these simple guidelines.



Kotlin mascot Kodee proportions

You can use Kodee in your digital and print materials. For this purpose, we have prepared a variety of Kotlin mascot assets for you to download and explore.

Download all assets



Kotlin mascot Kodee in action

Kotlin User Group brand assets

We provide Kotlin user groups with a logo that is specifically designed to be recognizable and convey a reference to Kotlin.

- The official Kotlin logo is associated with the language itself. It should not be used otherwise in different scopes, as this could cause confusion. The same applies to its close derivatives.
- User groups logo also means that the opinions and actions of the community are independent of the Kotlin team.
- Your opinions don't have to agree with ours, and we think this is the most beneficial model for a creative and strong community.

Download all assets

Style for user groups

Since the launch of the Kotlin community support program at the beginning of 2017, the number of user groups has multiplied, with around 2-4 new user groups joining us every month. Please check out the complete list of groups in the Kotlin User Groups section to find one in your area.

We provide new Kotlin user groups with a user group logo and a profile picture.

Branding image

There are two main reasons why we are doing it:

- Firstly, we received numerous requests from the community asking for special Kotlin style branded materials to help them be recognized as officially dedicated user groups.
- Secondly, we wanted to provide a distinct style for the user group and community content to make it clear which Kotlin-related materials are from the official team and which are created by the community.

Create the logo of your user group

To create a logo of your users group:

1. Copy the Kotlin user group logo file to your Google drive (you have to be signed in to your Google account).
2. Replace the Your City text with the name of your user group.
3. Download the picture and use it for the user group materials.

Belarusian Kotlin User Group sample

Belarusian Kotlin User Group Profile Picture sample

You can download a set of graphics including vector graphics and samples of cover pictures for social networks.

Create your group's profile picture for different platforms

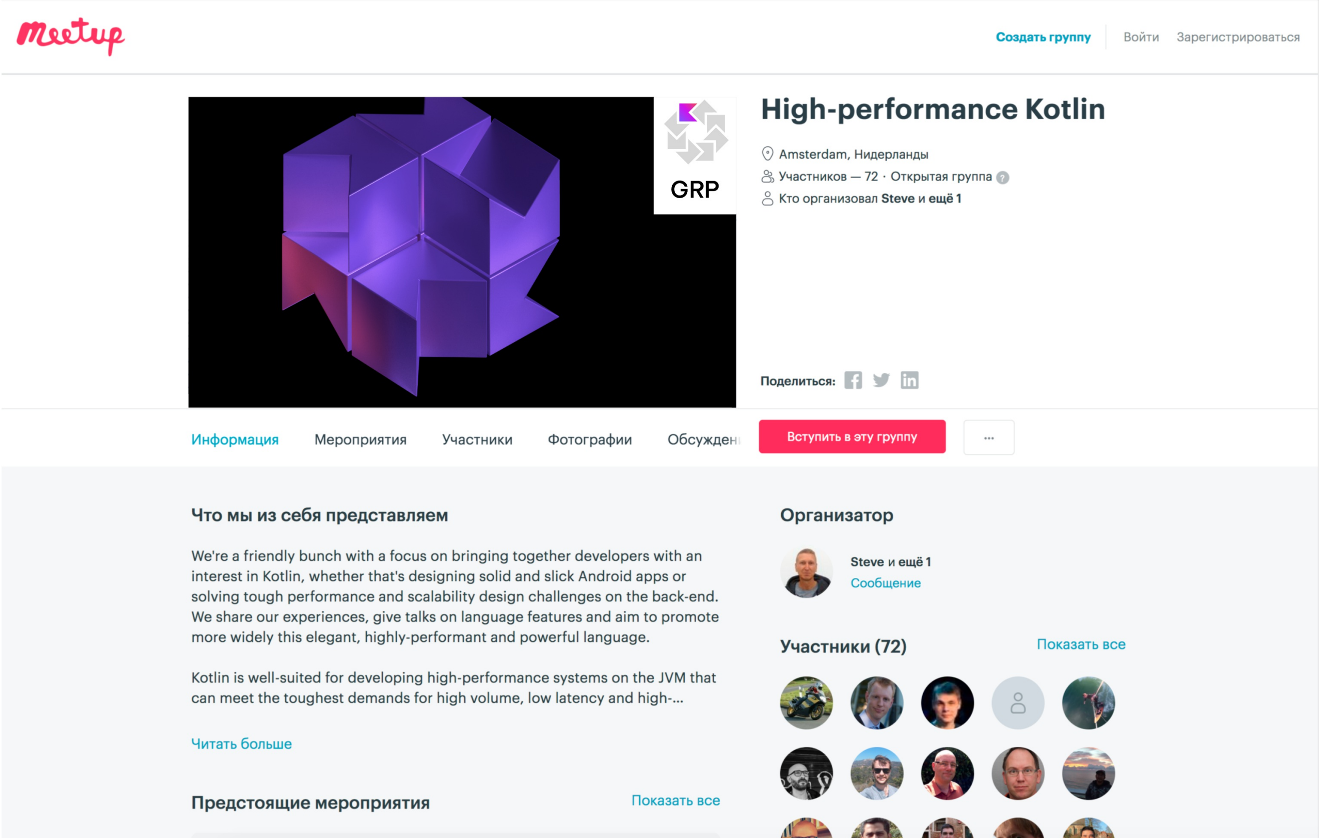
To create your group's profile picture:

1. Make a copy of the Kotlin user group profile picture file to your Google Drive (you have to be signed in to your Google account).
2. Add a shortened name of the user group's location (up to 4 capital symbols according to our default sample).
3. Download the picture and use it for your profiles on Facebook, Twitter, or any other platform.

Create meetup.com cover photo

To create a cover photo with a group's logo for meetup.com:

1. Make a copy of the picture file to your Google Drive (you have to be signed in to your Google account).
2. Add a shortened name of the user group's location to the logo on the right upper corner of the picture. If you want to replace the general pattern with a custom picture, click on the background pattern-picture, choose 'Replace Image', then 'Upload from Computer' or any other source.
3. Download the picture and use it for your profile on meetup.com.



User Group examples

Kotlin Night brand assets

JetBrains provides branding and materials for Kotlin Night events. Our team will prepare digital assets for the event promotion and ship your merchandise pack containing stickers and t-shirts. Check out what we have to make your Kotlin Night fun!

Download all assets

Social media

Stickers can be used to brand any media necessary for a Kotlin Night. Just stick them on anything you can get your hands on. It's fun!

Cover

Cover/Logo

Example of usage

Cover Social

Branding stickers

Stickers can be used to brand assets for a Kotlin Night. Just stick them on anything you can get your hands on. It is funny!

Example of usage

Stickers

Press-wall

You can decorate a press wall with stickers for unforgettable event pictures.

Example of usage

Press-wall

Press Wall

Sticky badges

Use stickers as badges for the attendees and boost networking at the event!

Board for stickers

Or you can provide a board where your guests can paste stickers with their impressions, feedback, and wishes.

board pack

T-shirts

Guests of the event are offered to paste stickers on the board with their impressions of the meeting. What does it mean for you?

Front print

Back print

Sticker pack