Kotlin In-Depth [Vol. II]

A Comprehensive Guide to Modern Multi-Paradigm Language

ALEKSEI SEDUNOV

bpb

Kotlin In-Depth [Vol. II]

A Comprehensive Guide to Modern Multi-Paradigm Language

ALEKSEI SEDUNOV

bpb

Kotlin In-Depth
[Vol. II]

A Comprehensive Guide to
Modern Multi-Paradigm Language

by

Aleksei Sedunov

FIRST EDITION 2020

ISBN: 978-93-89423-228

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

Tatiana, my guiding light
and the incessant source of inspiration

About the Author

Aleksei Sedunov has been working as a Java developer since 2008. After joining JetBrains in 2012 he’s been actively participating in the Kotlin language development focusing on IDE tooling for the IntelliJ platform. Currently he’s working in a DataGrip team, a JetBrains Database IDE, carrying on with using Kotlin as a main development tool.

Acknowledgement

Above all others I would like to give my gratitude to the entire Kotlin team at JetBrains which has created such a beautiful language and continues to relentlessly work on its improvement – especially Andrey Breslav who's been leading the language design from the very first day.

I'm really grateful to everyone at BPB publications for giving me this splendid opportunity for writing this book and lending a tremendous support in improving the text before it gets to the readers.

Last but not least I'd like to thank my beloved family for their support throughout the work on the book.

– Aleksei Sedunov

Preface

Since its first release in 2016 (and even long before that) Kotlin has been gaining in popularity as a powerful and flexible tool in a multitude of development tasks being equally well-equipped for dealing with mobile, desktop and server-side applications finally getting its official acknowledgment from Google in 2017 and 2018 as a primary language for Android development. This popularity is well-justified since language pragmatism, the tendency to choose the best practice among known solutions was one of the guiding principles of its design.

With the book you're holding in your hands I'd like to invite you to the beautiful world of Kotlin programming where you can see its benefits for yourself. After completing this book you'll have all the necessary knowledge to write in Kotlin on your own.

The second volume introduces you to the more advanced Kotlin features such as reflection, domain-specific languages and coroutines, discusses Java/Kotlin interoperability issues and explains how Kotlin can be used in various development areas, including testing, Android applications and Web. It's divided into the following 8 chapters:

Chapter 10 addresses the use of annotations which allow you to accompany Kotlin code with various metadata and explains the basics of Reflection API which provides access to runtime representation of Kotlin declarations.

Chapter 11 describes some advanced features which help developer in composing flexible APIs in a form of the domain-specific languages: operator overloading, delegated properties and builder-style DSLs based on the higher-order functions.

Chapter 12 discusses common issues of combining Java and Kotlin code within the same codebase and explain the specifics of using Java declarations in Kotlin code and vice versa.

Chapter 13 introduces the reader to the Kotlin coroutines library which inroduces a set of building blocks for programming asynchronous computations. Additionally it describes some utilities simplifying the use of Java concurrency API in Kotlin code.

Chapter 14 is devoted to the KotlinTest, a popular testing framework aimed specifically at the Kotlin developers. It describes various specification styles, explains the use of assertion API and addresses more advanced issues like using fixtures and test configurations.

Chapter 15 serves as an introduction to using Kotlin for development on the Android platform. It guides the reader through setting up an Android Studio project and explains basic aspects of Android development on an example of a simple calculator application.

Chapter 16 explains the basic features of the Ktor framework aimed at development of connected applications which make heavy use of Kotlin featutes and asynchronous computations.

Chapter 17 describes how to build a microservice application using Spring Boot and Ktor frameworks.

Downloading the code bundle and coloured images:

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/rrq2rot

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

CHAPTER 10

Annotations and Reflection

In this chapter, we will be covering two major topics. The first part will be devoted to annotations, which allow you to bind metadata to Kotlin declarations and access them later, at runtime. We will explain how to define and apply your own annotations and look at some built-in annotations, which affect the compilation of the Kotlin source code.

The second part will introduce us to the Reflection API, which defines a set of types comprising runtime representation of Kotlin declarations. We'll discuss how to obtain reflection objects, access their attributes and use callables, to dynamically invoke functions and properties.

Structure

- Defining and using annotation classes
- Built-in annotations
- Class literals and callable references
- Reflection API

Objective

Learn to apply annotations in Kotlin source code, as well as, declare your own annotation classes. Get an understanding of how to use Kotlin Reflection API to obtain runtime information about Kotlin declarations, and dynamically invoke functions and properties.

Annotations

Annotation is a special kind of Kotlin class that allows you to define custom metadata and bind them to the elements of your source code - declarations, expressions or whole files. Like their Java counterparts, Kotlin annotations

can be accessed at runtime. This ability is used extensively by various frameworks and processing tools, which rely on annotations for configuration and code instrumentation purposes.

Defining and using annotation classes

The syntax of annotation usage is rather similar to that of Java. The most basic case is annotating a declaration when you put a @-prefixed annotation name into its modifier list. For example, when using test framework, such as JUnit, you can mark test methods using annotation `@Test`:

```
class MyTestCase {
 @Test
 fun testOnePlusOne() {
  assert(1 + 1 == 2)
 }
}
```

Java vs. Kotlin: Unlike Java, some Kotlin annotations may also be applied to expressions. For example, the built-in `@Suppress` annotation can be used to suppress the compiler warnings for a particular expression in the source file:

```
val s = @Suppress("UNCHECKED_CAST") objects as List<String>
```

If you have multiple annotations for the same source file element, then you may group them inside square brackets:

```
@[Synchronized Strictfp] // the same as @Synchronized @Strictfp
fun main() { }
```

If you want to apply annotation to a primary constructor, then you need to use an explicit `constructor` keyword:

```
class A @MyAnnotation constructor ()
```

In *Chapter 4 (Working with Classes and Objects),* we have already used similar syntax to make a primary constructor private.

To define an annotation, you have to declare a class marked with a special `annotation` modifier:

```
annotation class MyAnnotation
@MyAnnotation fun annotatedFun() { }
```

Java vs. Kotlin: Please note the difference between the annotation

definitions in Kotlin and Java. While Java annotations have a syntactic form of an interface, Kotlin annotations comprise a special kind of class.

Unlike ordinary classes, annotation classes may not have members, secondary constructors or initializers:

```
annotation class MyAnnotation {
 val text = “???” // Error
}
```

However, since Kotlin 1.3, you can add nested classes, interfaces and objects (including companions) to the annotation body:

```
annotation class MyAnnotation {
 companion object {
  val text = “???”
 }
}
```

If you want to add custom attributes to your annotation, you may do so via constructor parameters. When an annotation like this is used, you need to provide the actual values for the parameters like a class constructor call:

```
annotation class MyAnnotation(val text: String)
@MyAnnotation(“Some useful info”) fun annotatedFun() { }
```

Please note, that the annotation parameters must always be marked as val.

Java vs. Kotlin: It is worth remembering that Java annotation attributes are specified in a form of parameter-less methods. However, in Kotlin, you use constructor parameters that also play the role of properties.

Similarly to ordinary constructors you may use default values and varargs:

```
annotation class Dependency(var arg val componentNames: String)
annotation class Component(val name: String = “Core”)

@Component(“I/O”)
class IO

@Component(“Log”)
@Dependency(“I/O”)
class Logger

@Component
@Dependency(“I/O”, “Log”)
class Main
```

Even though every Kotlin annotation is a kind of a class, you cannot

instantiate them the way you do with ordinary classes:

```
annotation class Component(val name: String = “Core”)
val ioComponent = Component(“IO”) // Error
```

Annotations can only be constructed using `@` syntax, as mentioned above. To retrieve an actual annotation instance (if it’s preserved at runtime), you may use Reflection API, which we will discuss in the upcoming sections.

Annotation classes cannot have explicit supertypes and cannot be inherited. They automatically inherit from the `Any` class and empty `Annotation` interface, which serves as a common supertype for all annotation classes.

Since annotation arguments are evaluated at the compilation time, you may not place arbitrary computations there. Furthermore, the compiler limits the range of possible types that you may use for annotation parameters:

- Primitive types such as Int, Boolean or Double;
- String;
- Enums;
- Other annotations;
- Class literals;
- Arrays of the types above.

Please note, that a parameter like this may not be nullable, because JVM does not allow you to store nulls in annotation attributes.

When you use another annotation as an argument, you do not have to put `@` prefix before its name. Instead, you write the annotation like an ordinary constructor call. Let us rework our previous example a little:

```
annotation class Dependency(vararg val componentNames: String)
 annotation class Component(
 val name: String = “Core”,
 val dependency: Dependency = Dependency()
)

@Component(“I/O”)
class IO

@Component(“Log”, Dependency(“I/O”))
class Logger

@Component(dependency = Dependency(“I/O”, “Log”))
```

```
class Main
```

Annotation parameters may have an explicit array type without using a `vararg`. When using an annotation like this, you may construct an array using the standard `arrayOf()` function:

```
annotation class Dependency(val componentNames: Array<String>)
@Component(dependency = Dependency(arrayOf(“I/O”, “Log”)))
class Main
```

Since Kotlin 1.2, you may also use a more concise syntax by enclosing the array elements inside square brackets:

```
annotation class Dependency(val componentNames: Array<String>)
@Component(dependency = Dependency([“I/O”, “Log”]))
class Main
```

Such array literals are currently only supported inside annotations.

Class literal gives you a representation of a class as a reflection object of type `KClass`. This type serves as a Kotlin counterpart of the `Class` type used in the Java language. The class literal consists of a class name followed by `::class`. Let us modify our component/depedency example to use class literals instead of names:

```
import kotlin.reflect.KClass

annotation class Dependency(vararg val componentClasses:
KClass<*>)
annotation class Component(
 val name: String = “Core”,
 val dependency: Dependency = Dependency()
)

@Component(“I/O”)
class IO

@Component(“Log”, Dependency(IO::class))
class Logger

@Component(dependency = Dependency(IO::class, Logger::class))
class Main
```

Java vs. Kotlin: Please note that instances of `java.lang.Class` may not be used in Kotlin annotations. However, during the JVM-targeted compilation Kotlin class literals are automatically converted into Java’s.

There are cases when a single declaration in a Kotlin source file corresponds

to multiple language elements that may have annotations. For example, suppose that we have the following class:

```
class Person(val name: String)
```

The `val` name: `String` in the code above serves as a shorthand declaration for a constructor parameter, a class property with getter and a backing field that is used to store a property value. Since each of those elements may have their own annotations, Kotlin allows you to specify a particular annotation target at its use site.

The use-site target is represented by a special keyword, which is placed before the annotation name and is separated by the : character. For example, if we want to place some annotation on a property getter, then we use the `get` keyword:

```
class Person(@get:A val name: String)
```

Most of the use-site targets are related to various components of a property. Such targets can be applied to any top-level or class property, as well as, the `val/var` parameter of a primary constructor:

- `property`: represents a property itself;
- `field`: represents the backing field (applicable only to properties that have a backing field);
- `get`: represents property getter;
- `set`: represents property setter (applicable only to mutable properties);
- `param`: represents the constructor parameter (applicable only to `val/var` parameters);
- `setparam`: represents the parameter of a property setter (applicable only to mutable properties);
- `delegate`: represents the field that stores a delegate object (applicable only to a delegated property, see *Chapter 11, Domain-Specific Languages*, for details);

The `get/set` targets allow you to annotate property accessors even when they are not explicitly present in your code (like `val` parameter in the example above). The same stands true for the `setparam` target, which has the same effect as the annotation setter parameter directly.

Annotations with use-site target can be also be grouped using the `[]` syntax. In this case, the target is applied to all of them. So, the definition

```
class Person(@get:[A B] val name: String)
```

is basically equivalent to:

```
class Person(@get:A @get:Bval name: String)
```

The receiver target applies annotation to a receiver parameter of the extension function or property:

```
class Person(val firstName: String, val familyName: String)
fun @receiver:APerson.fullName() = “$firstName $familyName”
```

Finally, the `file` target means that annotation is applied to the entire file. Such annotations must be placed at the beginning of the Kotlin file, before the import and package directives:

```
@file:JvmName(“MyClass”)
fun main() {
 println(“main() in MyClass”)
}
```

At runtime, file annotations are kept in the file facade class, which contains top-level functions and properties. In the *Chapter 12 (Java Interoperability)*, we shall discuss a group of file-level annotations (like the `@JvmName` above) that affect how such facade classes are visible from the Java code.

Now, we will look at some built-in annotations, which have a special meaning in the context of the Kotlin code.

Built-in annotations

Kotlin includes several built-in annotations, which have a special meaning in the context of the compiler. Some of them can be applied to the annotation classes themselves and allow you to specify options that would affect the usage of target annotations. Most of them serve as counterparts for similar meta-annotations available in Java language.

The `@Retention` controls how the annotation is stored. Like Java’s `@Retention` interface, you can choose from the three options represented by `AnnotationRetention` enum:

- `SOURCE`: This annotation exists at compile-time only and is not stored in

the compiler's binary output;

- `BINARY`: This annotation is stored in the compiler's output, but remains invisible for `Reflection` API;
- `RUNTIME`: This annotation is stored in the compiler's binary output and can be accessed via reflection.

By default, Kotlin annotations have `RUNTIME` retention, so that you do not have to worry about their availability via `Reflection` API. Please note, that, currently, the expression annotations cannot be preserved at runtime. So both, `BINARY` and `RUNTIME` retentions are forbidden for them:

```
@Target(AnnotationTarget.EXPRESSION)
annotation class NeedToRefactor // Error: must have SOURCE
retention
```

In a case like this, you have to specify the `SOURCE` retention explicitly:

```
@Target(AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
annotation class NeedToRefactor // Ok
```

Java vs. Kotlin: Please note, the difference between the default retention policies in Java and Kotlin. In Java, it is `RetentionPolicy.CLASS` (an equivalent of Kotlin's `AnnotationRetention.BINARY`), which means that the Java annotations are not available via reflection, unless you explicitly change their retention to `RUNTIME`.

`@Repeatable` specifies that the annotation can be applied to the same element more than once:

```
@Repeatable
@Retention(AnnotationRetention.SOURCE)
annotation class Author(val name: String)
@Author("John")
@Author("Harry")
class Services
```

By default, annotations are not repeatable, and if you try to apply a non-repeatable annotation multiple times, the compiler will report an error:

```
@Deprecated("Deprecated")
@Deprecated("Even more deprecated") // Error: non-repeatable
annotation
class OldClass
```

Please note, that currently, repeatable annotations cannot be kept at runtime and hence, must have an explicit `SOURCE` retention.

`@MustBeDocumented` specifies that the annotation must be included into the documentation, since it is considered to be a part of the public API. This annotation plays the same role as Java's `@Documented` and is supported by Dokka, the standard Kotlin documentation engine (the way `@Documented` is supported by Javadoc tool).

The `@Target` indicates what kind of language elements are supported by the annotation. The possible kinds are specified as a `vararg` of constants from the `AnnotationTarget` enum:

- `CLASS`: Any class, interface or object, including annotation classes themselves;
- `ANNOTATION_CLASS`: Any annotation class; this effectively allows you to define your own meta-annotations;
- `TYPEALIAS`: Any type alias definition;
- `PROPERTY`: Any property, including the `val/var` parameters of a primary constructor (but not local variables);
- `FIELD`: The backing field of a property;
- `LOCAL_VARIABLE`: Local variables only (excluding parameters);
- `VALUE_PARAMETER`: Parameters of constructors, functions and property setters;
- `CONSTRUCTOR`: Primary and secondary constructors only;
- `FUNCTION`: Functions including lambdas, and anonymous functions (but not constructors or property accessors);
- `PROPERTY_GETTER/PROPERTY_SETTER`: Property getters/setters only;
- `FILE`: The annotation can be applied to an entire file;
- `TYPE`: Any type specification, like the type of a variable, parameter or return value of a function;
- `EXPRESSION`: Any expression;

The `TYPE_PARAMETER` constant is reserved for a future use, but currently it is not supported. As a consequence, you cannot apply annotations to type parameters of generic declarations yet.

When `@Target` is not specified, the annotation can be applied to any language element except type alias, type parameter, type specification, expression and file. For example, if you want your annotation to be applicable to a file, you have to specify it explicitly.

Java vs. Kotlin: The `AnnotationTarget` class is quite similar to the `ElementType` enum found in the JDK. Please note, the difference though, between their `TYPE` constants - in Kotlin, `AnnotationType.TYPE` refers to type specification (which corresponds to `ElementType.TYPE_USAGE` in Java), while `ElementType.TYPE` means actual declaration of class or interface (similarly, to `AnnotationTarget.CLASS`).

Also, please note, that, unlike Java, Kotlin does not support package-level annotations (thus no counterpart for `ElementType.PACKAGE`). However, you can define the annotation at the level of the source file. In the *Chapter 12 (Java Interoperability)*, we will see how file annotations can be used to tune Java-Kotlin interoperability.

The following annotations are equivalent to corresponding Java modifiers:

- `@Strictfp`: Restrict the precision of floating-point operations for better portability between different platforms;
- `@Synchronized`: Force annotated function or property accessor, to acquire/release the monitor before/after executing the body;
- `@Volatile`: Make updates of the annotated backing field immediately visible to other threads;
- `@Transient`: Indicate that the annotated field is ignored by default serialization mechanism;

Since `@Synchronized` and `@Volatile` are related to concurrency support, we will defer their detailed treatment till the *Chapter 13 (Concurrency)*.

The `@Suppress` annotation allows you to suppress some compiler warnings specified by their internal names. This annotation may be applied to any target, including expressions and files. For example, you can use it to disable spurious warnings related to casts when you are certain that your code is valid:

```
val strings = listOf<Any>("1", "2", "3")
val numbers = listOf<Any>(1, 2, 3)
// No warning:
```

```
val s = @Suppress("UNCHECKED_CAST") (strings as List<String>)[0]
// Unchecked cast warning:
val n = (numbers as List<Number>)[1]
```

The annotation affects all the code inside the element that it is applied to. For example, you can suppress all the warnings inside a particular function:

```
@Suppress("UNCHECKED_CAST")
fun main() {
 val strings = listOf<Any>("1", "2", "3")
 val numbers = listOf<Any>(1, 2, 3)
 val s = (strings as List<String>)[0] // No warning
 val n = (numbers as List<Number>)[1] // No warning
 println(s + n) // 12
}
```

Or inside the entire file, if you use `@Suppress` with the file use-site target:

```
@file:Suppress("UNCHECKED_CAST")

val strings = listOf<Any>("1", "2", "3")
val numbers = listOf<Any>(1, 2, 3)

fun takeString() = (strings as List<String>)[0] // No warning
fun takeNumber() = (numbers as List<Number>)[1] // No warning

@Suppress("UNCHECKED_CAST")
fun main() {
 println(takeString() + takeNumber()) // 12
}
```

IDE Tips: There is no need to look up warning names or memorize them, as IntelliJ can insert @Suppress annotations automatically. To do so, press Alt + Enter while your caret is placed within the warning region (see Figure 10.1) and choose one of the Suppress… actions from the Annotator submenu. These actions are applicable to warnings reported by IDE inspections as well.

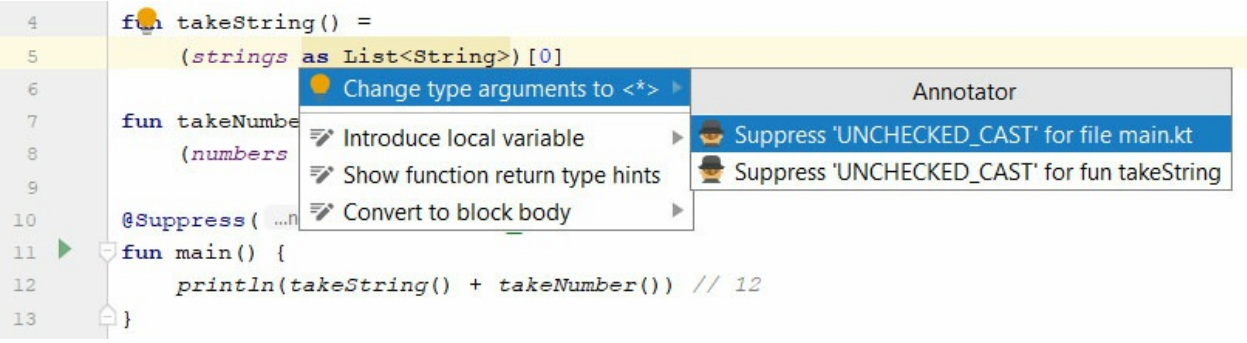

Figure 10.1: *Suppressing compiler warnings*

Another useful annotation, `@Deprecated`, is very similar to its Java counterpart. When you mark some declaration as deprecated, the client code is discouraged from using it. In IDE, the usages of deprecated declarations are displayed in strike-through font (as shown on Figure 10.2). When using `@Deprecated`, you specify a message that usually clarifies why this declaration is deprecated and/or what the user should use instead.

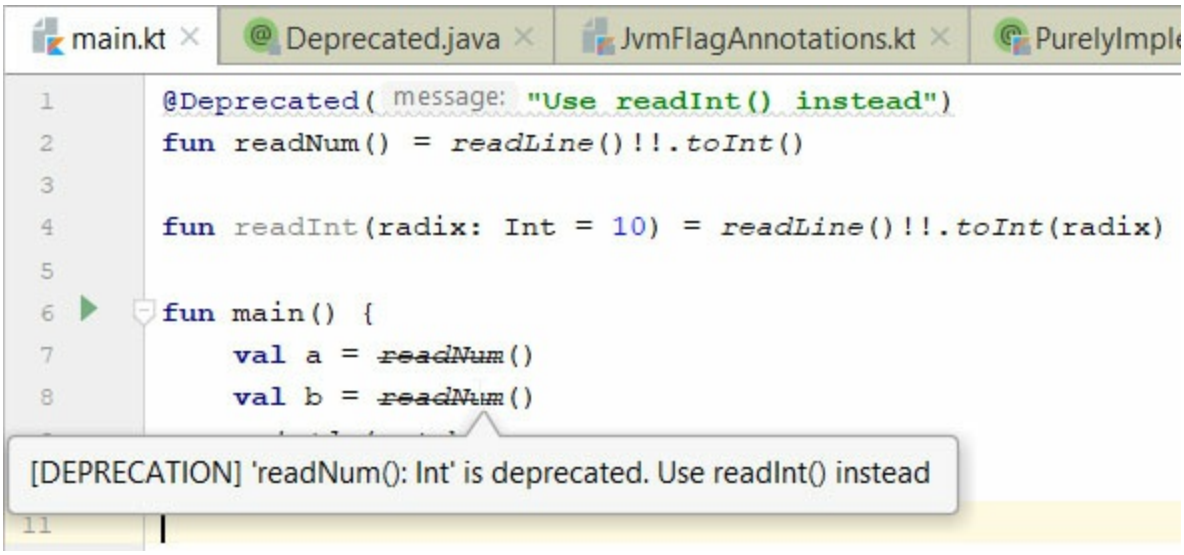

Figure 10.2: *Deprecated declaration*

Unlike Java, `@Deprecated` in Kotlin provides additional features. To begin with, you can specify a string with a replacement expression. In this case, deprecated usages can be automatically changed into the desired form by using the corresponding quick-fix from the Alt + Enter menu (see an example in Figure 10.3). Let's suppose that we want to replace `readNum()` with `readInt()` above. In that case, we can write:

```
@Deprecated(
  “Use readInt() instead”, // Message
  ReplaceWith(“readInt()”) // Replacement expression
)
fun readNum() = readLine()!!.toInt()
```

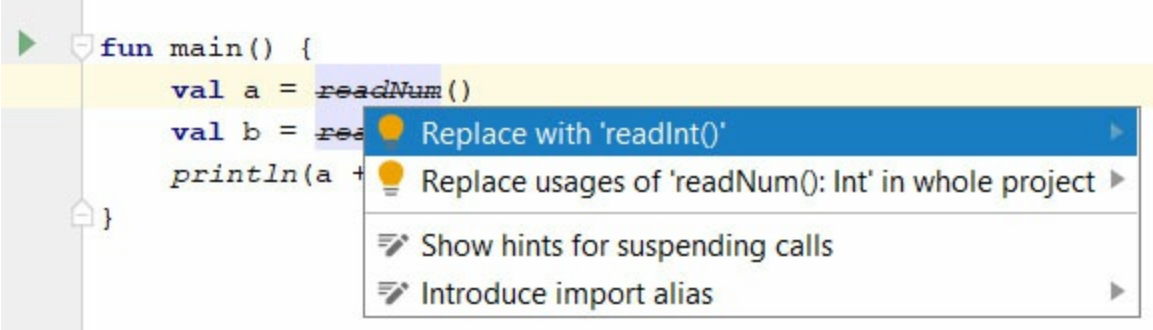

Figure 10.3: *Using quick fix to replace a deprecated usage*

Please note that `ReplaceWith` is an annotation too. That is why you can place it inside the `@Deprecated` usage. However, `@ReplaceWith` cannot be used by itself. Take a look at its definition:

```
@Target()
@Retention(BINARY)
@MustBeDocumented
public annotation class ReplaceWith(
 val expression: String,
 var arg val imports: String
)
```

As you can see, it has no supported targets. It can only be constructed as a part of another annotation like `@Deprecated`.

The additional `vararg` parameter of `ReplaceWith` allows you to specify a list of the necessary imports to add on the replacement. This is useful if the replacement code refers to declarations from the non-default/non-current package.

Another feature is an ability to choose the severity of deprecation, which is represented by `DeprecationLevel` enum:

- `WARNING`: Usages of deprecated declaration are reported as warnings; this is default behavior;
- `ERROR`: Usages of deprecated declarations are reported as compilation errors;
- `HIDDEN`: Deprecated declaration can’t be accessed at all;

Using deprecation levels allows you to implement a smooth deprecation policy, which is relevant for team development. First you deprecate the declaration with default level, so that its existing usages are reported as

warnings. This gives the developers time to replace them, without breaking the compilation of the code. Then, you raise the deprecation error to `ERROR`, thus prohibiting the newly introduced usages of the deprecated code. When you make sure that no one is going to use this code again, you may safely drop it from your code base.

Figure 10.4 shows an example of using the `ERROR` level to forbid using `readNum()` function:

```
@Deprecated(
        message: "Use readInt() instead",
        ReplaceWith( expression: "readInt()"),
        DeprecationLevel.ERROR
)
fun readNum() = readLine()!!.toInt()

fun readInt(radix: Int = 10) = readLine()!!.toInt(radix)

fun main() {
    val a = readNum()
    val b = readNum()
    println(a + b)
}
```

Figure 10.4: *Deprecated declaration with ERROR level*

Some built-in annotations like `@Throws`, `@JvmName`, or `@NotNull` are used for tuning the Java/Kotlin interoperability. We will cover them in the *Chapter 12 (Java Interoperability)*.

Reflection

Reflection API is a set of types, functions and properties, which gives you an access to runtime representations of classes, functions and properties. This is useful when your code has to work with classes that are not available at the compile time, but still conform to some common contract. For example, you may load classes dynamically as plugins and call their members knowing their signature.

In the following section, we will discuss what are the elements that comprise the Kotlin Reflection API and give examples of their use.

Java vs. Kotlin: Please note, that the Kotlin Reflection is not self-sufficient. In some cases, like class search and loading, we have to rely on the facilities provided by the Java Reflection API. When it comes to manipulating the Kotlin-specific aspects of your code (like properties or objects), using the Kotlin API gives you a more concise and idiomatic way to access them at runtime.

Reflection API overview

The Reflection classes reside in the `kotlin.reflect` package and can be loosely divided into two basic groups: callables, which deal with representation of properties and function (including constructors), and classifiers, which provide a runtime representation of classes and type parameters. The diagram in Figure 10.5 gives an overview of the basic reflection types.

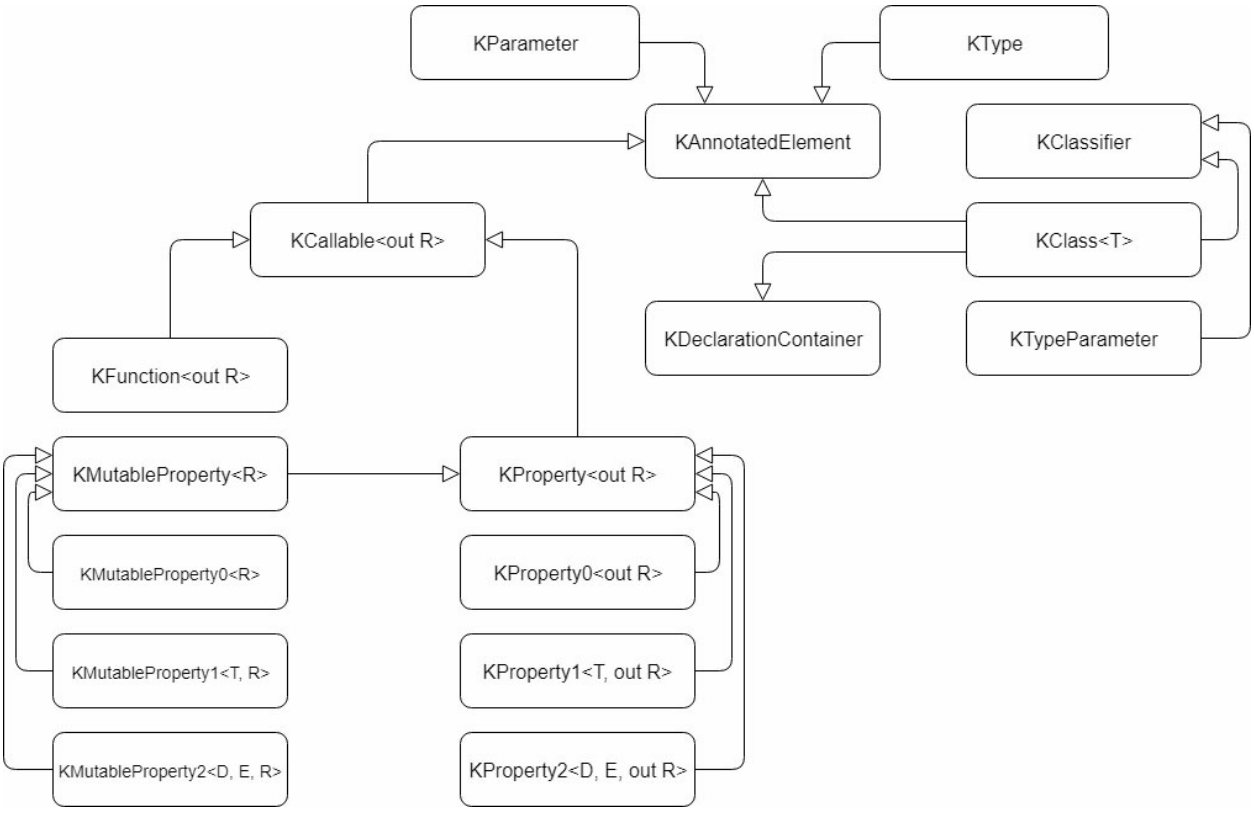

Figure 10.5: *Basic reflection types*

All the reflection types are descendants of the `KAnnotatedElement`, which allows you to access the annotations defined for a specific language element,

such as function, property or class. The `KAnnotatedElement` has a single property that returns a list of annotation instances:

```
public val annotations: List<Annotation>
```

Let's revisit our earlier example with the `@Component/@Dependency` annotations:

```
import kotlin.reflect.KClass

annotation class Dependency(vararg val componentClasses:
KClass<*>)
annotation class Component(
 val name: String = "Core",
 val dependency: Dependency = Dependency()
)

@Component("I/O")
class IO

@Component("Log", Dependency(IO::class))
class Logger

@Component(dependency = Dependency(IO::class, Logger::class))
class Main
```

Let us suppose that we want to retrieve the annotations associated with the `Main` class. We can do so by using the annotations property on its class literal:

```
fun main() {
 val component = Main::class.annotations
 .filterIsInstance<Component>()
 .firstOrNull() ?: return
 println("Component name: ${component.name}")
 val depText = component.dependency.componentClasses
 .joinToString { it.simpleName ?: "" }
 println("Dependencies: $depText")
}
```

If you run the code above, then you will get:

```
Component name: Core
Dependencies: IO, Logger
```

In the following sections, we will consider the API of more specific types related to classifiers and callables.

Classifiers and types

In terms of the Kotlin Reflection, a classifier is a declaration that defines a type. Such declarations are represented by the `KClassifier` interface, which has two more specific varieties currently:

- `KClass<T>`, which represents a declaration of some class, interface or object with the compile-time type `T`;
- `KTypeParameter`, which represents a type parameter of some generic declaration.

Please note, that currently, type aliases have no representation in the Reflection API. This, in particular, means that even though you may apply annotations to type aliases, such annotations cannot be retrieved at runtime. Type alias support is expected to be added in a future release of Kotlin.

Since `KClassifier` doesn't define its own members, let's go straight to the specifics of classes and type parameters.

There are two basic ways to obtain an instance of `KClass`. The first is to use a class literal syntax that we have already seen in the sections about annotations:

```
println(String::class.isFinal) // true
```

Apart from classes, this syntax is also supported for the reified type parameters. In *Chapter 9 Generics*, we have mentioned that type parameters of generic inline functions may be reified. This means that the compiler substitutes the actual types instead of them, while inlining the function body at its call site. Let's, for example, define the `cast()` function:

```
inline fun <reified T>Any.cast() = this as? T
```

Suppose that we then call in a following way:

```
val obj: Any = "Hello"
println(obj.cast<String>())
```

Behind the scene the compiler will actually generate a code:

```
val obj: Any = "Hello"
println(obj as? String)
```

You may also use the `::class` syntax an for arbitrary expression to obtain a

runtime class of its resulting value:

```
println((1 + 2)::class) // class kotlin.Int
println(“abc”::class) // class kotlin.String
```

Another way to get a `KClass`, is to use the `kotlin` extension property to convert an instance of the `java.lang.Class`. This is useful in finding a class dynamically, by its qualified name. Since the Kotlin Reflection does not have its own class search API yet, it has to rely on a platform-specific one:

```
valstringClass = Class.forName(“java.lang.String”).kotlin
println(stringClass.isInstance(“Hello”)) // true
```

The opposite conversion is given by the `java` extension property:

```
println(String::class.java) // class java.lang.String
```

Let’s now look at the `KClass` API. The first group of the `KClass` members allow you to determine if a class of interest has a specific modifier:

```
val isAbstract: Boolean
val isCompanion: Boolean
val isData: Boolean
val isFinal: Boolean
val isInner: Boolean
val isOpen: Boolean
val isSealed: Boolean
```

Another property from the same group, `visibility`, gives you a visibility level of a class declaration as an instance of the `KVisibility` enum:

```
enum class Kvisibility {
 PUBLIC,
 PROTECTED,
 INTERNAL,
 PRIVATE
}
```

Please note, that visibility may be null if it cannot be denoted in the Kotlin source code: for example, when `KClass` represent a local class.

The next group of properties allows you to retrieve a class name:

```
val simpleName: String?
val qualifiedName: String?
```

The `simpleName` property returns a simple name, which was used in its source code. When a class has no name (e.g. a class representing an object

expression), the result is null.

Similarly, the `qualifiedName` property gives you a qualified name of a class, which includes a full name of the containing package. When a class is local or nested into local, the result is null, since such classes cannot be used from the top level. Hence, they have no qualified name. The same goes for classes that have no name in the source code.

You can also use the `jvmName` extension property, which gives you a qualified name of a class from Java's point of view. This name may differ from the one given by the `qualifiedName`. Some built-in Kotlin types do not have their own JVM representation and rely on existing Java classes instead. For example, the Any class does not exist as a separate Java class; for the Java code, it is the same as `java.lang.Object`:

```
println(Any::class.qualifiedName) // kotlin.Any
println(Any::class.jvmName) // java.lang.Object
```

The `isInstance()` function allows you to check if a given object is an instance of the class represented by its receiver. This function works like the `is` operator, when applied to a non-nullable type:

```
println(String::class.isInstance("")) // true
println(String::class.isInstance(12)) // false
println(String::class.isInstance(null)) // false
```

The next group of `KClass` properties provides access to its member declarations:

- `constructors`: Collection of both primary and secondary constructors as instances of the `KFunction` type;
- `members`: Collection of member functions and properties represented by the `KCallable` instances, including all members inherited from the supertypes;
- `nestedClasses`: Collection of nested classes and objects, including companions;
- `typeParameters`: The list of type parameters represented by the `KTypeParameter` type (when the class in question is not generic, the list is empty).

For example, in the following code, we use reflection to dynamically create an instance of the `Person` class, and then call its `fullName()` function:

```
class Person(valfirstName: String, val familyName: String) {
 fun fullName(familyFirst: Boolean): String = if (familyFirst) {
  “$familyName $firstName”
 } else {
  “$firstName $familyName”
 }
}
fun main() {
 val personClass = Class.forName(“Person”).kotlin
 val person = personClass.constructors.first().call(“John”,
 “Doe”)
 val fullNameFun = personClass.members.first { it.name ==
 “fullName” }
 println(fullNameFun.call(person, false)) // John Doe
}
```

When `KClass` represents object declarations, the `constructors` property will always return an empty collection. To obtain an actual instance, you may use the `objectInstance` property:

```
object O {
 val text = “Singleton”
}
fun main() {
 println(O::class.objectInstance!!.text) // Singleton
}
```

When `KClass` does not represent an object, `objectInstance` is null.

Finally, for a sealed class (`isSealed == true`), you can also get a list of all direct inheritors via the `sealedSubclasses` property.

One more piece of information that you can get from a `KClass` is given by the `supertypes` property, which returns a list of `KType` instances. We will get to the `KType` API a little later, but for now let’s consider a simple example:

```
open class GrandParent
open class Parent :GrandParent()
interface IParent
class Child : Parent(), IParent
fun main() {
 println(Child::class.supertypes) // [Parent, IParent]
}
```

Please note, that the `supertypes` property returns only the immediate supertypes (thus `GrandParent` is absent in the output above), so if you want to access the indirect ones, you will have to perform an inheritance graph

traversal.

Another classifier variety is represented by the `KTypeParameter` interface. Compared to the `KClass`, it is rather simple and provides only four properties:

```
val isReified: Boolean
val name: String
val upperBounds: List<KType>
val variance: KVariance
```

The `upperBounds` give you a list of upper bound types, like the supertypes property of `KClass`. The list is never empty, as every type parameter has a bound (which is `Any?` by default). There may also be more than one bound, if the type parameter is used in the type constraint. For example:

```
interface MyMap<K : Any, out V>
fun main() {
 val parameters = MyMap::class.typeParameters
 // K: [kotlin.Any], V: [kotlin.Any?]
 println(parameters.joinToString { "${it.name}:
 ${it.upperBounds}" })
}
```

The `variance` property returns a constant of the `KVariance` enum, which represents the declaration-site variance of a type parameter:

```
enum class KVariance{ INVARIANT, IN, OUT }
```

Now, let's look at how types are represented in the Kotlin Reflection through the `KType` interface. A Kotlin type is characterized by the following aspects:

- Nullability given by the `isMarkedNullable` property, which distinguishes, say, `List<String>` and `List<String>?`;
- classifier (given by the `classifier` property), which refers to the class, interface or object declaration defining the type. For example, the `List` part of `List<String>`;
- The list of type arguments given by the arguments property: `<String>` for `List<String>`, `<Int, Boolean>` for `Map<Int, Boolean>`, and so on.

The type argument is represented by `KTypeProjection` interface which contains information about the type itself as well its use-site variance:

```
val type: kotlin.reflect.KType?
val variance: kotlin.reflect.KVariance?
```

Both the properties return null for the star projection *.

This wraps up our overview of the classifier types. In the following section we will focus on the callable part of Reflection API.

Callables

The notion of a callable unites the properties and functions that you can call to obtain some result. In the Reflection API, they are represented by a generic interface `KCallable<out R>`, where `R` denotes either the return type of a function or a type of property.

A way to get an instance of `KCallable` is to use the callable references that we have discussed in *Chapter 5 Leveraging Advanced Functions and Functional Programming*:

```
fun combine(n: Int, s: String) = "$s$n"
fun main() {
 println(::combine.returnType) // kotlin.String
}
```

You can also access the member functions and properties via the corresponding `KClass` instance. However, please note, that, currently, the Reflection API does not allow you to obtain top-level callables in this manner.

Let's now look at the common members defined in `KCallable` itself. Like `KClass`, we have a group of properties, which allow you to check the presence of certain modifiers:

```
val isAbstract: Boolean
val isFinal: Boolean
val isOpen: Boolean
val isSuspend: Boolean
val visibility: KVisibility?
```

We have not yet come across the `suspend` modifier corresponding to the `isSuspend` property. This modifier is used to define the callables that support suspendable computations. In *Chapter 13 (Concurrency)*, we will discuss this issue in more detail.

The next group of properties represent the signature of a property or a function:

```
val name: String
```

```
val typeParameters: List<KTypeParameter>
val parameters: List<KParameter>
val returnType: KType
```

Please note, that for members and extensions, the first parameter is reserved for a receiver. When the callable is a member and an extension at the same time, the second parameter is reserved as well. For example:

```
import kotlin.reflect.KCallable
val simpleVal = 1
val Int.extValget() = this
class A {
 val Int.memberExtValget() = this
}
```

```
fun main() {
 fun printParams(callable: KCallable<*>) {
  println(
   callable.parameters.joinToString(prefix = “[“, postfix = “]”)
   {
   it.type.toString()
   }
  )
 }
 // []
 printParams(::simpleVal)
 // [kotlin.Int]
 printParams(Int::extVal)
 // [A, kotlin.Int]
 printParams(A::class.members.first { it.name == “memberExtVal”
 })
}
```

The `KParameter` interface contains information about the function/constructor parameter or receiver(s) of a member/extension declaration:

```
val index: Int
val isOptional: Boolean
val isVararg: Boolean
val name: String?
val type: KType
```

The `isOptional` property returns true when the parameter has a default value; the value itself is currently not available via reflection. Please note, that the parameter name may be null if its name is not available or was not present in the source code. The latter holds true for the parameters representing receiver values as well.

The `kind` property indicates whether the `KParameter` corresponds to an ordinary value, or a dispatch/extension receiver. It can return one of the constants defined in the `KParameter.Kindenum`:

- `INSTANCE`: Dispatch receiver of member declaration;
- `EXTENSION_RECEIVER`: Extension receiver of extension declaration;
- `VALUE`: Ordinary parameter.

`KCallable` also defines the `call()` member that allows you to dynamically invoke the backing callable:

```
fun call(vararg args: Any?): R
```

In the case of a function-based callable, the `call()` invokes the function itself. If the callable corresponds to a property, the getter is used instead. We have already seen an example of using `call()` to invoke constructor and member function. Let's look at a property example, as applied to the same `Person` class:

```
fun main() {
 val person = Person("John", "Doe")
 val personClass = person::class
 val firstName = personClass.members.first { it.name ==
 "firstName" }
 println(firstName.call(person)) // John
}
```

An alternative `callBy()` function allows you to pass arguments in the form of a map:

```
fun callBy(args: Map<KParameter, Any?>): R
```

Let's now move to the more specialized callable kinds. The `KProperty` interface adds checks for property-specific modifiers:

```
val isConst: Boolean
val isLateinit: Boolean
```

You may also access the property getter as an instance of the `KFunction` type:

```
val myValue = 1
fun main() {
 println(::myValue.getter()) // 1
}
```

The `KMutableProperty` extends the `KProperty` by adding a setter:

```
var myValue = 1
fun main() {
 ::myValue.setter(2)
 println(myValue) // 2
}
```

The `KProperty` also has subtypes - `KProperty0`, `KProperty1` and `KProperty2`, representing the properties without a receiver, with a single receiver (either dispatch, or extension) and with a pair of receivers, (member extension) respectively. These subtypes refine the types of getters by making them implement the corresponding functional type. That feature allowed us to use `::myValue.getter` as a function in the example mentioned above. Similar subtypes, with refined setters, are defined for the `KMutableProperty` as well.

The final reflection type that we are going to consider is the `KFunction`, which quite expectedly, represents functions and constructors. The only members added to this interface are related to function-specific modifiers checks:

```
val isInfix: Boolean
val isInline: Boolean
val isOperator: Boolean
val isSuspend: Boolean
```

The `isInfix` and `isOperator` checks are related to the operator functions, which we will cover in *Chapter 11 (Domain-Specific Languages)*.

Please note, that the `KFunction` by itself is not an inheritor of any functional type, since it can represent functions with different arity. However, some functional types may be implemented by more specific subtypes of the `KFunction`. We have already seen that through the example of accessors defined in `KProperty0/KProperty1/KProperty2`. Another important case is the callable references, which always conform to the proper functional type. For example:

```
import kotlin.reflect.KFunction2
fun combine(n: Int, s: String) = "$s$n"
fun main() {
 val f: KFunction2<Int, String, String> = ::combine
 println(f(1, "2")) // 12
}
```

As you can see, the callable reference in this example has a `KFunction2<Int,`

`String, String>` type which is a subtype of `(Int, String) -> String`. However, please note, that unlike `KProperty0` and similar other types, `KFunction0/KFunction1/…` only exist during compilation. At runtime they are represented by the synthetic classes, similar to the ones created for lambdas.

One more thing worth noting is the ability to access callables with restricted visibility. In some cases, you may need to reflectively call a private function. In Java, an attempt to do this may produce an exception, so in general you have to make a reflection object accessible by calling `setAccessible(true)` beforehand. In Kotlin, you use the `isAccessible` property for the same purpose:

```
import kotlin.reflect.KProperty1
import kotlin.reflect.jvm.isAccessible

class SecretHolder(private val secret: String)

fun main() {
 val secretHolder = SecretHolder(“Secret”)
 val secretProperty = secretHolder::class.members
 .first { it.name == “secret” } as KProperty1<SecretHolder,
 String>
 secretProperty.isAccessible = true
 println(secretProperty.get(secretHolder))
}
```

Conclusion

This chapter has brought us to the topics of annotations and reflection. We have discussed how to annotate pieces of the Kotlin code and obtain the associated metadata at runtime. We went through major built-in annotations and explained how to define your own annotation classes. We have also introduced you to the Kotlin Reflection API. Now you are familiar with how to access the attributes of both classifiers and callables, and use them in a dynamic fashion.

In the next chapter, we are going to address the subject of designing your own APIs in a way that resembles the domain-specific languages, bringing an air of declarative programming into your codebase.

Questions

1. How to define a new annotation? Compare the Kotlin annotation syntax with that of Java.
2. How are annotations used in the Kotlin code?
3. Which built-in annotations are available in Kotlin?
4. What is annotation use-site target? How is it related to targets specified by the @Target meta-annotation?
5. What are the basic types comprising the Kotlin Reflection API?
6. Describe class literal and callable reference syntax.
7. Describe KClass API. How to convert between KClass and Java's Class instances?
8. Describe KCallable API.

CHAPTER 11

Domain-Specific Languages

Domain-specific language (DSL) is a language tailored for a specific function or domain. Such languages are heavily used in software development to deal with various tasks such as, describing software configurations, test specification, workflow rules, UI design, data manipulation and so on. The main advantage of DSL is its simplicity: instead of relying on low-level constructs of general-purpose languages such as Java, you use a domain-specific primitive, thus dealing with a task in its own terms. However, the approach has some drawbacks as it can be difficult to embed a DSL code into a general-purpose program, because they are written in different languages. As a result, DSL programs are usually stored outside its host code or simply embedded into string literals, which complicates the compile-time validation and code assistance in IDE.

Kotlin, however, can offer a solution to you. In this chapter, we will address a set of features, which allow you to design DSLs that can be seamlessly combined with the rest of the Kotlin code. The idea is to design a special API, which can be used in way that resembles the domain-specific language. So while your code may look like it is written in a different language, it still remains a valid Kotlin code. In other words, you get the advantages of both, the domain-specific approach and the power of compiled languages, including strong type-safety guarantees.

Structure

- Operator overloading
- Delegated properties
- Higher-order functions and DSLs

Objective

Learn advanced features of Kotlin, which help the developer in designing an API in the form of domain-specific languages.

Operator overloading

Operator overloading is a languages feature, which allows you to assign the custom meaning to built-in Kotlin operators like +, -, *, /, etc. In the previous chapters, we have already seen how + semantics vary depending on the type of values it applies to: arithmetic sum for numbers, concatenation for strings, appending an element for collection and so on. This is possible because + is overloaded, that is, it has many different implementations.

In Kotlin, operator expression is just a syntactic sugar over function calls. To implement an operator, you just need to define an extension or member function following a certain convention and mark it with the operator keyword. For example, by defining:

```
operator fun String.times(n: Int) = repeat(n)
```

We extend * operator (corresponding to the `times()` function) to `String/Int` pairs, which in turn allows us to write:

```
println(“abc” * 3) // abcabcabc
```

Since the operators are backed by some functions, we can always replace them by ordinary calls. For example, the code above has the same meaning as:

```
println(“abc”.times(3))
```

Even the built-in operations like addition, can be written in this form, although in case of primitive types, operations like addition or subtraction are optimized by the compiler to avoid actual function calls for the sake of performance:

```
val x = 1.plus(2) // the same as 1 + 2
```

IDE Tips: IntelliJ plugin can convert the explicit call of the operator function into corresponding unary/binary expression. To do that, you just need to press Alt + Enter on the operator token or function name and choose an appropriate conversion action. *Figure 11.1* shows an example of conversion like this, which replaces times() call with binary *.

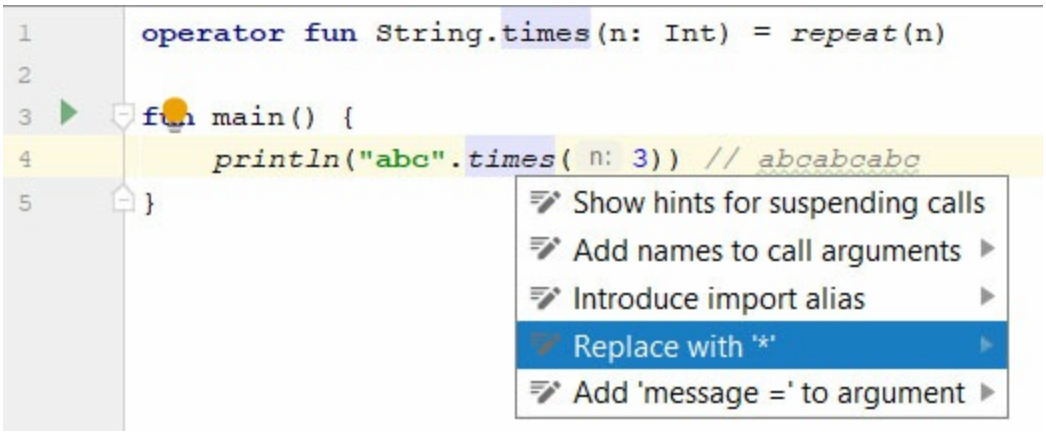

Figure 11.1: *Converting explicit call to an operator form*

In the following section, we will talk about the conventions related to various Kotlin operators and consider some examples of their implementation.

Unary operations

Overloadable unary operators include the prefix +, - and !. When you use such operators, the compiler automatically unfolds them into a call of an appropriate function (*Table 11.1*):

Expression	Meaning
+e	e.unaryPlus()
-e	e.unaryMinus()
!e	e.not()

Table 11.1: *Unary operator conventions*

The functions may either be members, or extensions, defined for the type of the argument expression. They may not have any parameters and their result type becomes the type of the whole unary expression.

For example, consider an enum class that represents basic RGB colors, as well as their combinations:

```
enum class Color {
  BLACK, RED, GREEN, BLUE, YELLOW, CYAN, MAGENTA, WHITE
}
```

Using the `not()` convention we can introduce the ! operator as a shorthand for complementary colors:

```
enum class Color {
 BLACK, RED, GREEN, BLUE, YELLOW, CYAN, MAGENTA, WHITE;
 operator fun not() = when (this) {
  BLACK -> WHITE
  RED -> CYAN
  GREEN -> MAGENTA
  BLUE -> YELLOW
  YELLOW -> BLUE
  CYAN -> RED
  MAGENTA -> GREEN
  WHITE -> BLACK
 }
}

fun main() {
 println(!Color.RED) // CYAN
 println(!Color.CYAN) // RED
}
```

By defining operator functions as extensions, you can support the respective operations for expressions of arbitrary types. For example:

```
operator fun <T> ((T) -> Boolean).not(): (T) -> Boolean = {
!this(it) }
```

Using the function above, we can now apply the ! operator to any single-parameter predicate:

```
fun isShort(s: String) = s.length<= 4
fun String.isUpperCase() = all { it.isUpperCase() }

fun main() {
 val data = listOf("abc", "abcde", "ABCDE", "aBcD", "ab")
 println(data.count(::isShort)) // 3
 println(data.count(!::isShort)) // 2
 println(data.count(String::isUpperCase)) // 1
 println(data.count(!String::isUpperCase)) // 4
}
```

Increments and decrements

Increment (++) and decrement (—) operators can be overloaded by providing parameter-less functions, `inc()` and `dec()`, for the corresponding operand type. The return value of these functions must correspond to the next and previous value respectively, according to some ordering. The way `inc()/dec()` are used, depends on whether the operator is written in its

prefix or postfix form, similar to the way ++/-- work for numbers. For example, let us suppose that we have an enum class listing rainbow colors:

```
enum class RainbowColor {
 RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET;
}
```

Let's define `inc()/dec()` according to the ordering above, looping around the first and last elements so that the next of `VIOLET` is `RED` and the prior of `RED` is `VIOLET`:

```
enum class RainbowColor {
 RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET;
 operator fun inc() = values[(ordinal + 1) % values.size]
 operator fun dec() = values[(ordinal + values.size - 1) %
 values.size]
 companion object {
  private val values = enumValues<RainbowColor>()
 }
}
```

Now, let's consider how increment and decrement would work for this class. As we have already seen in the *Chapter 2 (Language Fundamentals)*, the postfix form of ++/-- operators update a variable, but returns its value before the change. This is also true for overloaded operators. Consider, for example, the following code:

```
var color = RainbowColor.INDIGO
println(color++)
```

In effect it means:

```
var color = RainbowColor.INDIGO
val _oldColor = color
color =color.inc()
println(_oldColor) // INDIGO
```

In case of the prefix form, the result of the increment/decrement expression is equal to the updated value, so the below fragment:

```
var color = RainbowColor.INDIGO
println(color++)
```

would actually translate into:

```
var color = RainbowColor.INDIGO
color = color.inc()
```

```
println(color) // VIOLET
```

Please note, that the presence of an assignment like `color = color.inc()` implies two things:

- ++ and -- are only applicable to mutable variables;
- The return type of `inc()/dec()` functions must be a subtype of their receiver type.

Binary operations

Kotlin allows you to overload most of the binary operators. Like the unary ones, you need to provide a corresponding operator function. The major difference is that the binary operator functions take their left operand as the receiver, while the right operand is passed as an ordinary argument.

Table 11.2 lists the conventional names for arithmetic operators, .. and `in/!in`.

Expression	Meaning
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)
a in b	b.contains(a)
a !in b	!b.contains(a)

Table 11.2: *Binary operator conventions*

Initially, the % operation was a shorthand for the `mod()` operator function, which is currently superseded by `rem()`. As of now, the `mod()` convention is still available but deprecated.

For example, let’s consider a simple prototype implementation of rational numbers supporting basic arithmetic operations:

```
import kotlin.math.abs
class Rational private constructor(
```

```
 val sign: Int,
 val num: Int,
 val den: Int
) {
 operator fun unaryMinus() = Rational(-sign, num, den)
 operator fun plus(r: Rational): Rational {
  val gcd = gcd(den, r.den)
  val newDen = den/gcd*r.den
  val newNum = newDen/den*num*sign + newDen/r.den*r.num*r.sign
  val newSign = newNum.sign()
  return Rational(newSign, abs(newNum), newDen)
 }
 operator fun minus(r: Rational) = this + (-r)

 operator fun times(r: Rational): Rational {
  return of(sign*r.sign*num*r.num, den*r.den)
 }

 operator fun div(r: Rational): Rational {
  return of(sign*r.sign*num*r.den, den*r.num)
 }

 override fun toString(): String {
  return "${sign*num}" + if (den != 1) "/$den" else ""
 }

 companion object {
  private fun Int.sign() = when {
   this > 0 -> 1
   this < 0 -> -1
   else -> 0
  }
  private tailrec fun gcd(a: Int, b: Int): Int {
   return if (b == 0) a else gcd(b, a % b)
  }
  fun of(num: Int, den: Int = 1): Rational {
   if (den == 0) throw ArithmeticException("Denominator is
   zero")
   val sign = num.sign() * den.sign()
   val numAbs = abs(num)
   val denAbs = abs(den)
   val gcd = gcd(numAbs, denAbs)
   return Rational(sign, numAbs/gcd, denAbs/gcd)
  }
 }
}
```

Using operator conventions, we can build arithmetic operations out of

`Rational` instances:

```
fun r(num: Int, den: Int = 1) = Rational.of(num, den)
fun main() {
 // 1/2 - 1/3
 println(r(1, 2) - r(1, 3)) // 1/6
 // 2/3 + (1/3)/2
 println(r(2, 3) + r(1, 3)/r(2)) // 5/6
 // 3/4 * 8/9 / (2/3)
 println(r(3, 4)*r(8, 9)/r(2, 3)) // 1
 // (1/10)*2 - 2/6
 println(r(1, 10)*r(2) - r(2, 6)) // -2/15
}
```

We can also introduce some additional operator functions, which would allow us to mix `Rational` objects with values of other types such as `Int`. For example:

```
operator fun Rational.plus(n: Int) = this + Rational.of(n)
operator fun Int.plus(r: Rational) = r + this

operator fun Rational.minus(n: Int) = this - Rational.of(n)
operator fun Int.minus(r: Rational) = Rational.of(this) - r

fun main() {
 // -1/3 + 2
 println(r(-1, 3) + 2) // 5/3
 // 1 - (1/4)*(1/2)
 println(1 - r(1, 4)*r(1, 2)) // 7/8
}
```

To demonstrate usage of`..` operation let’s define `RationalRange` class representing a closed interval between two rational numbers:

```
class RationalRange(val from: Rational, val to: Rational) {
 override fun toString() = "[$from, $to]"
}
```

Now we can define the `rangeTo()` function, which would construct an instance of `RationalRange`:

```
operator fun Rational.rangeTo(r: Rational) = RationalRange(this,
r)

fun main() {
 println(r(1, 4)..r(1)) // [1/4, 1]
}
```

The `in/!in` operations are expressed by the `contains()` operator function. Please note, that unlike all other binary operations, arguments of `contains()` are swapped as compared to its operator form. Let's enhance the `RationalRange` class with an ability to check if a given number belongs to the range:

```
private fun Rational.isLessOrEqual(r: Rational): Boolean {
 return num*r.den<= r.num*den
}

class RationalRange(val from: Rational, val to: Rational) {
 override fun toString() = "[$from, $to]"

 operator fun contains(r: Rational): Boolean {
  return from.isLessOrEqual(r) &&r.isLessOrEqual(to)
 }

 operator fun contains(n: Int) = contains(r(n))
}

fun main() {
 // 1/2 in [1/4, 1]
 println(r(1, 2) in r(1, 4)..r(1)) // true

 // 1 not in [5/4, 7/4]
 println(1 !in r(5, 4)..r(7, 4)) // true
}
```

One more group of overloadable operators deals with comparisons like < and >. These operators do not correspond to separate functions, instead you use a single `compareTo()` function to implement the full set of comparisons for a given combination of operand types. This function returns an Int value, which signifies the comparison result. All comparison operations are implemented on top of it according to the Table 11.3:

Expression	Meaning
a < b	a.compareTo(b) < 0
a <= b	a.compareTo(b) <= 0
a > b	a.compareTo(b) > 0
a >= b	a.compareTo(b) >= 0

Table 11.3: *Comparison operator conventions*

Now we can get rid of `isLessThan()` function above, replacing it with a more

general `compareTo()` implementation:

```
operator fun Rational.compareTo(r: Rational): Int {
 val left = num * r.den
 val right = r.num * den
 return when {
  left < right -> -1
  left > right -> 1
  else -> 0
 }
}
operator fun Rational.compareTo(n: Int) = compareTo(r(n))
operator fun Int.compareTo(r: Rational) = -r.compareTo(this)
class RationalRange(val from: Rational, val to: Rational) {
 override fun toString() = "[$from, $to]"
 operator fun contains(r: Rational) = r >= from && r <= to
 operator fun contains(n: Int) = contains(r(n))
}
fun main() {
 println(1 > r(1, 3)) // false
 println(r(3/4) <= r(7/8)) // true
}
```

One more binary convention that we have already used in the previous chapters is concerned with equality. When you use == or !=, the compiler automatically reduces the operator to the `equals()` call. Please note, that the `equals()` implementation does not need an explicit operator modifier since it is inherited from the base version declared in the Any class. It is for the same reason that `equals()` can only be implemented as a member; even if you define `equals()` as an extension it won't be used in place of a `==/!=` operators, since extensions are always shadowed by member declarations with the same signature.

Please note, that Kotlin does not allow you to overload && and ||: they are built-in operations supported only for Boolean values. The same holds true for Kotlin identity equality operations, === and `!==`.

What if you want to implement a binary operation with a custom name? Although Kotlin doesn't allow you to introduce new operators, you can use ordinary identifiers as names for infix operations. We'll see how to do this in the next section.

Infix operations

We have already seen operations like to or until, which can be used as infix operations:

```
val pair1 = 1 to 2 // infix call
val pair2 = 1.to(2) // ordinary call
```

To enable such calls, you need to mark a function with an infix modifier. Like binary operators, the function of interest must be either a member, or an extension and have a single parameter. For example, that is how the standard to function is defined:

```
infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

Let’s refine our earlier predicate example a little by introducing infix operations for predicate conjunction and disjunction:

```
infix fun <T> ((T) -> Boolean).and(
 other: (T) -> Boolean
): (T) -> Boolean {
 return { this(it) && other(it) }
}

infix fun <T> ((T) -> Boolean).or(
 other: (T) -> Boolean
): (T) -> Boolean {
 return { this(it) || other(it) }
}
```

Now we can use them to combine the functional literals in a more concise manner:

```
fun main() {
 val data = listOf("abc", "abcde", "ABCDE", "aBcD", "ab")
 println(data.count(::isShort and String::isUpperCase)) // 0
 println(data.count(::isShort or String::isUpperCase)) // 4
 println(data.count(!::isShort or String::isUpperCase)) // 2
 println(data.count(!(::isShort and String::isUpperCase))) // 5
}
```

Please bear in mind, that all infix operations have the same precedence. For example, complex expressions involving the and/or operations that we have defined above, would parse differently than similar expressions with built-in || and &&boolean operators. For example, the expression:

```
!::isShort or String::isEmpty and String::isUpperCase
```

would essenially mean:

```
(!::isShort or String::isEmpty) and String::isUpperCase
```

While the boolean expression:

```
!s.isShort() || s.isEmpty() &&s.isUpperCase()
```

would be equivalent to:

```
!s.isShort() || (s.isEmpty() &&s.isUpperCase())
```

This is due to && having a higher precedence than ||.

Assignments

The next group of binary operations deals with augmented assignments like +=. In *Chapter 7, Exploring Collections and I/O*, we have seen that these operations behave differently for mutable and immutable collections. in other words, applying += to a variable of an immutable collection type would create a new collection object and assign it to the variable, thus changing its value. The variable must be defined as mutable in this case:

```
var numbers = listOf(1, 2, 3)
numbers += 4
println(numbers) // [1, 2, 3, 4]
```

However, when using += on a mutable collection, we modify the collection content while preserving an original object identity:

```
val numbers = mutableListOf(1, 2, 3)
numbers += 4
println(numbers) // [1, 2, 3, 4]
```

Please note, that if we put a mutable collection into a mutable variable, then += would produce an error, since the compiler cannot decide which convention to follow:

```
var numbers = mutableListOf(1, 2, 3)
// Should we update a variable or collection content?
numbers += 4 // Error
println(numbers)
```

Both the conventions can be supported for arbitrary types, owing to the respective operator functions. The behavior of augmented assignments depends on the following factors (see Table 11.4):

- The presence of the corresponding binary operator function: `plus()` for

+=, `minus()` for -= and so on;

- The presence of the custom assignment function: `plusAssign()` for `+=`, `minusAssign()` for -= and so on;
- Mutability of assignment left-hand side.

Expression	Meaning	
	Simple assignment reduction	**Custom assignment function**
a += b	**a = a.plus(b)**	**a.plusAssign(b)**
a -= b	**a = a.minus(b)**	**a.minusAssign(b)**
a *= b	**a = a.times(b)**	**a.timesAssign(b)**
a /= b	**a = a.div(b)**	**a.divAssign(b)**
a %= b	**a = a.rem(b)**	**a.remAssign(b)**

Table 11.4: *Assignment operator conventions*

Let's consider the possible cases. When left-hand side has a corresponding binary operator (e. g. `plus()`) but not a custom assignment function, augmented assignment is reduced to a simple one. This happens with primitive types and immutable collections. We can also use such assignments for our Rational objects, since they already support binary operations like + and -. For example:

```
var r = r(1, 2) // ½
// The same as r = r + r(1, 3)
r += r(1, 3) // 1/2 + 1/3
println(r) // 5/6
```

Please note, that the assignment left-hand side must be a mutable variable in this case.

When a left-hand side has only a custom assignment function (e.g. `plusAssign()`, but not `plus()`), then an augmented assignment is reduced to its call. To illustrate this convention, let's revisit the TreeNode class that we have introduced in *Chapter 9 (Generics)* and change its API a little:

```
class TreeNode<T>(val data: T) {
 private val _children = arrayListOf<TreeNode<T>>()

 var parent: TreeNode<T>? = null
 private set
```

```
  operator fun plusAssign(data: T) {
    val node = TreeNode(data)
    _children += node
    node.parent = this
  }

  operator fun minusAssign(data: T) {
    val index = _children.indexOfFirst { it.data == data }
    if (index < 0) return
    val node = _children.removeAt(index)
    node.parent = null
  }

  override fun toString() =
    _children.joinToString(prefix = "$data {", postfix = "}")
}
```

Now we can use += and -= operator on instances of `TreeNode` to add and remove the tree elements:

```
val tree = TreeNode("root")
tree += "child 1"
tree += "child 2"
println(tree) // root {child 1 {}, child 2 {}}
tree -= "child 2"
println(tree) // root {child 1 {}}
```

Please note, that the custom assignment functions must have the Unit return type.

When the left-hand side has both, a custom assignment and a simple binary operation, the result depends on the mutability of the left-hand side:

- If the left-hand side is immutable, then the compiler chooses the custom assignment function because simple assignment is not applicable;
- If the left-hand side is a mutable variable, then the compiler reports an error, because it results in ambiguity: whether a += b is supposed to mean a = a + b or `a.plusAssign(b)`.

The behavior mentioned above, is demonstrated by the Kotlin mutable collection classes such as lists or sets, because they have both the plus()/minus() functions inherited from immutable collections and their own plusAssign()/minusAssign().

Invocations and indexing

The invocation convention allows you to use values in call expressions like functions. To do this, you just need to define the `invoke()` function with the necessary parameters. Values of functional types automatically get `invoke()` as their member, but you can also add invocation support to arbitrary type. Suppose, for example, that we have the following function:

```
operator fun <K, V> Map<K, V>.invoke(key: K) = get(key)
```

Then we can use any Map instance as a function that returns a value by its key:

```
val map = mapOf("I" to 1, "V" to 5, "X" to 10)
println(map("V")) // 5
println(map("L")) // null
```

A useful case is to add the `invoke()` function to the companion object, turning it into a factory. For example, if we augment out `Rational` class with an extension:

```
operator fun Rational.Companion.invoke(num: Int, den: Int = 1) =
of(num, den)
```

We can construct Rational instances by referring to the class name:

```
val r = Rational(1, 2)
```

The code above may look like a direct constructor call, but it, in fact, reduces to the invocation chain: `invoke()`→`of()`→ private constructor of Rational.

A similar convention allows you to use the indexing operator [] similar to how it applies to strings, arrays, lists and maps. The underlying call depends on whether the indexing expression is used as a value or a left-hand side of the assignment. In the first case, the compiler assumes reading access and reduces the indexing operator to the call of the `get()` function with the same set of arguments.

```
val array = arrayOf(1, 2, 3)
println(array[0]) // the same as println(array.get(0))
```

However, when the indexing expression is used as an assignment left-hand side, the compiler reduces it to the call of the `set()` function, which on top of indices takes the assigned value as its last argument:

```
val array = arrayOf(1, 2, 3)
array[0] = 10 // the same as array.set(0, 10)
```

Indices are not necessarily integers; in fact, they may be arbitrary values. For example, the indexing operator for maps takes the key value as its argument.

For example, let's add both the `get()/set()` operators to our `TreeNode` class to access its children:

```
class TreeNode<T>(var data: T) {
 private val _children = arrayListOf<TreeNode<T>>()

 var parent: TreeNode<T>? = null
 private set

 operator fun plusAssign(data: T) {
  val node = TreeNode(data)
  _children += node
  node.parent = this
 }

 operator fun minusAssign(data: T) {
  val index = _children.indexOfFirst { it.data == data }
  if (index < 0) return
  val node = _children.removeAt(index)
  node.parent = null
 }
 operator fun get(index: Int) = _children[index]
 operator fun set(index: Int, node: TreeNode<T>) {
  node.parent?._children?.remove(node)
  node.parent = this
  _children[index].parent = null
  _children[index] = node
 }
}
fun main() {
 val root = TreeNode("Root")
 root += "Child 1"
 root += "Child 2"
 println(root[1].data) // Child 2
 root[0] = TreeNode("Child 3")
 println(root[0].data) // Child 3
}
```

A more sophisticated case is using the indexing operator in an augmented assignment. The resulting code depends on the meaning of the assignment operator for the type of left-hand side, which is effectively the return type of the `get()` operator function. For example, if we consider an array of `Rational` objects that do not have the `plusAssign()` function, the code:

```
val array = arrayOf(r(1, 2), r(2, 3))
array[0] += Rational(1, 3)
would mean:
val array = arrayOf(r(1, 2), r(2, 3))
array[0] = array[0] + r(1, 3)
Or, reducing everything to function calls:
val array = arrayOf(r(1, 2), r(2, 3))
array.set(0, array.get(0) + r(1, 3))
```

However, if we use an array of `TreeNode` objects that have the `plusAssign()` function, but no `plus()`, a similar fragment:

```
val array = arrayOf(TreeNode("Root 1"), TreeNode("Root 2"))
array[0] += TreeNode("Child 1")
```

would translate to:

```
val array = arrayOf(TreeNode("Root 1"), TreeNode("Root 2"))
array.get(0).plusAssign(TreeNode("Child 1"))
```

Please note, that the get() function is required in both the cases.

Destructuring

We have already seen how to use destructuring declarations for instances of data classes to declare multiple variables at once and initialize them to the values of corresponding data class properties. By using operator overloading you can enable this feature for arbitrary types. All you need is to define a parameter-less member/extension function `componentN(,)` where N is a 1-based number. Then each entry in a destructuring declaration initialized by an instance of corresponding receiver type is assigned a value returned by the component function with the respective index.

To demonstrate this convention, let's define the component functions for the `RationalRange` class that we have introduced in an earlier section:

```
operator fun RationalRange.component1() = from
operator fun RationalRange.component2() = to
```

Now we can apply destructuring to our `RationalRange` instances:

```
fun main() {
 val (from, to) = r(1, 3)..r(1, 2)
 println(from) // 1/3
 println(to) // 1/2
}
```

Data classes are no different in this regard. It is just that their component functions are autogenerated by the compiler rather than written explicitly. The Kotlin standard library includes some extension component functions as well. That is what allows you to destructure map entries:

```
val map = mapOf("I" to 1, "V" to 5, "X" to 10)
for ((key, value) in map) {
 println("$key = $value")
}
```

Or extract first elements of a list or an array:

```
val numbers = listOf(10, 20, 30, 40, 50)
val (a, b, c) = numbers
println("$a, $b, $c") // 10, 20, 30
```

Iteration

In *Chapter 3 (Defining Functions)*, we have introduced the for loop statement, which can be applied to various objects including strings, ranges and collections. Their common feature that allows us to use for loop, is the presence of the `iterator()` function, which returns the corresponding Iterator instance. By defining this function as either a member, or an extension, you can support iteration via for statement for any type you like.

As an example, let's support iteration for the `TreeNode` class that we have introduced in an earlier section:

```
operator fun <T>TreeNode<T>.iterator() = children.iterator()
```

Now we can use `TreeNode` instances in a for loop without explicit references to its children member. Consider the program:

```
fun main() {
 val content = TreeNode("Title").apply {
  addChild("Topic 1").apply {
   addChild("Topic 1.1")
   addChild("Topic 1.2")
  }
  addChild("Topic 2")
  addChild("Topic 3")
 }
 for (item in content) {
  println(item.data)
 }
```

```
}
```

Running it would print:

```
Topic 1
Topic 2
Topic 3
```

This concludes our discussion of operator overloading in Kotlin. In the next section, we will talk about the delegation mechanism, which allows you to introduce new kinds of properties into the Kotlin code.

Delegated properties

Delegated properties give you a way to implement the custom property access logic hidden behind a simple syntactic facade. We have already seen an example of the lazy delegate, which defers property computation till its first access:

```
val result by lazy { 1 + 2 }
```

Conciseness of the property delegates makes them a helpful tool in designing both simple-to-use APIs and domain-specific languages.

Like the operators that we have discussed in the previous sections, implementation of delegated property is based on a set of conventions, which allows you to define how a property is read or written and control the construction of the delegate object itself. In this section, we will talk about these conventions in more detail as well as address some ready-to-use delegates provided by the Kotlin standard library.

Standard delegates

The Kotlin standard library includes a bunch of ready-to-use delegate implementations, which cover many common use cases. In *Chapter 4 (Working with Classes and Objects)*, we have already seen an example of a delegate like this, representing a lazy property:

```
val text by lazy { File("data.txt").readText() }
```

In fact, the `lazy()` function has three versions that allow you to fine-tune the behavior of a lazy property in a multi-threaded environment. By default, it creates a thread-safe implementation, which uses synchronization to

guarantee that the lazy value is always initialized by a single thread; in this case the delegate instance also serves as a synchronization object.

When necessary, you can also specify your own synchronization object using another `lazy()` version:

```
private val lock = Any()
val text by lazy(this) { File("data.txt").readText() }
```

You can also choose between 3 basic implementations, by passing a value of the `LazyThreadSafetyMode` enum:

- `SYNCHRONIZED`: Property access is synchronized so that only one thread can initialize its value (this implementation is used by default);
- `PUBLICATION`: Property access is synchronized in such a way that the initializer function can be invoked multiple times, but only the result of its first call becomes the property value;
- `NONE`: Property access is not synchronized; in a multi-threaded environment, the property behavior is effectively undefined

The major difference between `SYNCHRONIZED` and `PUBLICATION` becomes apparent, if the initializer function has side effects. For example, if we have a property like this:

```
val myValue by lazy {
 println(“Initializing myValue”)
 123
}
```

The message gets printed once, at the most, because the `SYNCHRONIZED` mode (which is used by default) ensures that the initializer is not called multiple times. However, if we change the safety mode to `PUBLICATION`:

```
val myValue by lazy(LazyThreadSafetyMode.PUBLICATION) {
 println("Initializing myValue")
 123
}
```

The property values remain the same, but the message is printed as many times as there are threads that are trying to initialize `myValue`.

The `NONE` mode provides the fastest implementation and is useful when you can guarantee that the initializer is never called by more than one thread. A common case is a `lazy` local variable:

```
fun main() {
 val x by lazy(LazyThreadSafetyMode.NONE) { 1 + 2 }
 println(x) // 3
}
```

Please note, that if the initializer throws an exception, the property remains uninitialized. So the delegate will try to reinitialize it on the next access attempt.

Some standard delegates can be constructed by the members of `kotlin.properties.Delegates` object. The `notNull()` function provides a delegate that allows you to defer property initialization:

```
import kotlin.properties.Delegates.notNull
var text: String by notNull()
fun readText() {
 text = readLine()!!
}
fun main() {
 readText()
 println(text)
}
```

The semantics of the `notNull()` delegate is basically the same as that of `lateinit` properties: internally the null value is used as a marker of the uninitialized property, so that if it still happens to be null when you try to read from it, the delegate throws NPE. In most situations, it is worth using `lateinit` properties instead of `notNull()`, since `lateinit` has more concise syntax and better performance. An exception is a case of primitive types, which are not supported by `lateinit`:

```
import kotlin.properties.Delegates.notNull
var num: Int by notNull() // Can’t use lateinit here
fun main() {
 num = 10
 println(num) // 10
}
```

The `observable()` function allows you to define a property that sends a notification when its value is changed. It takes an initial value, a lambda, which is invoked after each change:

```
import kotlin.properties.Delegates.observable
class Person(name: String, val age: Int) {
 var name: String by observable(name) { property, old, new ->
  println("Name changed: $old to $new")
```

```
  }
}
fun main() {
  val person = Person("John", 25)
  person.name = "Harry" // Name changed: John to Harry
  person.name = "Vincent" // Name changed: Harry to Vincent
  person.name = "Vincent" // Name changed: Vincent to Vincent
}
```

Please note, that a notification is sent even if the new value is the same as the old one. The lambda should check this by itself, if necessary.

The `vetoable()` function constructs a similar delegate, but takes a lambda, which returns a Boolean and is called before actual modification. If this lambda returns false, the property value remains unchanged:

```
import kotlin.properties.Delegates.vetoable

var password: String by vetoable("password") { property, old, new
->
  if (new.length< 8) {
    println("Password should be at least 8 characters long")
    false
  } else {
    println("Password is Ok")
    true
  }
}
fun main() {
  password = "pAsSwOrD" // Password is accepted
  password = "qwerty" // Password should be at least 8 characters
  long
}
```

If you want to combine both, pre-and post-change notifications, then you may implement your own delegate by subclassing the `ObservableProperty` and overriding the `beforeChange()/afterChange()` functions.

Standard library also allows you to store/retrieve the property value using a map, where the property name serves as a key. You can do this by using a map instance as a delegate:

```
class CartItem(data: Map<String, Any?>) {
  val title: String by data
  val price: Double by data
  val quantity: Int by data
}
```

```
fun main() {
 val item = Cartitem(mapOf(
  "title" to "Laptop",
  "price" to 999.9,
  "quantity" to 1
 ))
 println(item.title) // Laptop
 println(item.price) // 999.9
 println(item.quantity) // 1
}
```

When you access a property, its value is taken from a map and cast down to the expected type. `Map` delegates should be used with care because they break type safety: in particular, access to a property value will fail with cast exception, if it does not contain a value of the expected type.

Using this feature, you can also define mutable variables backed by a mutable map:

```
class CartItem(data: MutableMap<String, Any?>) {
 var title: String by data
 var price: Double by data
 var quantity: Int by data
}
```

What if standard delegates are not enough? In such a case, you implement your own by following language conventions. We will see how to do this, in the next section.

Creating custom delegates

To create your own property delegate, you need a type that defines special operator function(s), which implement the reading and writing of the property value. The reader function must be named `getValue` and have two parameters:

1. `receiver`: Contains the receiver value and must be the same type as a receiver of the delegated property (or its supertype);
2. `property`: Contains the reflection object representing a property declaration, it must be of the type `KProperty<*>` or its supertype.

Parameter names are not actually important, only their types matter. The return type of the `getValue()` function must be the same as the type of the

delegated property (or its subtype).

For example, let’s create a delegate which memorizes property value, associating it with a particular receiver to create a kind of cache:

```
import kotlin.reflect.KProperty

class CachedProperty<in R, out T : Any>(val initializer: R.() ->
T) {
 private valcachedValues = HashMap<R, T>()

 operator fun getValue(receiver: R, property: KProperty<*>): T {
  return cachedValues.getOrPut(receiver) {
  receiver.initializer() }
 }
}
fun <R, T : Any> cached(initializer: R.() -> T) =
CachedProperty(initializer)

class Person(val firstName: String, val familyName: String)
valPerson.fullName: String by cached { "$firstName $familyName" }
fun main() {
 val johnDoe = Person("John", "Doe")
 val harrySmith = Person("Harry", "Smith")
 // First access for johnDoe receiver, computed and stored to
 cache
 println(johnDoe.fullName)
 // First access for harrySmith receiver, computed and stored to
 cache
 println(harrySmith.fullName)
 // Repeated access for johnDoe receiver, taken from cache
 println(johnDoe.fullName)
 // Repeated access for harrySmith receiver, taken from cache
 println(harrySmith.fullName)
}
```

Since `fullName` is a top-level property, its delegate becomes part of the global state, and the property value is only initialized once for a particular receiver (if we set multi-threading issues aside).

The `ReadOnlyProperty` interface from the `kotlin.properties` package can serve as a good starting point for creating custom, read-only delegates. This interface defines an abstract version of the `getValue()` operator, which you will need to implement in your own class:

```
interface ReadOnlyProperty<in R, out T> {
 operator fun getValue(thisRef: R, property: KProperty<*>): T
}
```

For a read-write delegate, which can be applied to var properties, you also need to define the corresponding `setValue()` function that is invoked upon each property assignment. This function must have a `Unit` return type and take three parameters:

1. `receiver`: has the same meaning as for `getValue()`;
2. `property`: has the same meaning as for `getValue()`;
3. `newValue`: the new value of a property, which must have the same type as a property itself (or its supertype).

In the following example, we define a delegate class, which implements a final version of `lateinit` property, which doesn't allow initializing it more than once:

```
import kotlin.reflect.KProperty
class FinalLateinitProperty<in R, T : Any> {
 private lateinit var value: T

 operator fun getValue(receiver: R, property: KProperty<*>): T {
  return value
 }

 operator fun setValue(receiver: R,
property: KProperty<*>,
newValue: T) {
   if (this::value.isInitialized) throw IllegalStateException(
    "Property ${property.name} is already initialized"
   )
   value = newValue
 }
}
fun <R, T : Any>finalLateInit() = FinalLateinitProperty<R, T>()
var message: String by finalLateInit()
fun main() {
 message = "Hello"
 println(message) // Hello
 message = "Bye" // Exception: Property message is already
 initialized
}
```

The Kotlin standard library also includes a mutable version of the `ReadOnlyProperty` interface, which is called `ReadWriteProperty`. Similarly, you can implement this interface in your delegate class:

```
public interface ReadWriteProperty<in R, T> {
```

```
 operator fun getValue(thisRef: R, property: KProperty<*>): T
 operator fun setValue(thisRef: R, property: KProperty<*>,
 value: T)
}
```

Please note, that the `getVersion()/setVersion()` functions may be defined as either members, or extensions. The latter option allows you to turn any object into a delegate of some kind. Delegation to the `Map/MutableMap` instances, in particular, is implemented by the extension function from the Kotlin standard library:

```
inline operator fun <V, V1 : V> Map<in String, V>.getValue(
 thisRef: Any?,
 property: KProperty<*>
): V1 {…}
```

Since Kotlin 1.1, you can control the delegate instantiation via the `provideDelegate()` function. By default, the delegate instance is defined by the expression coming after the `by` keyword in a property declaration. Alternatively, you can also pass an intermediate instance that serves as a delegate factory of some kind, with the `provideDelegate()` function. Like `getValue()`, this function takes the property receiver and reflection object as parameters, instead of retrieving the property value returns an actual delegate object. This can be useful when the delegate needs the property metadata for proper initialization.

Let us suppose that we want to introduce the `@NoCache` annotation, which prevents property caching. In this case, we would like our `CachedProperty` implementation to throw the exception early, during property initialization, rather than deferring failure till the moment the property is accessed. We can achieve this by adding the delegate provider, which validates the target property before creating the delegate:

```
@Target(AnnotationTarget.PROPERTY)
annotation class NoCache

class CachedPropertyProvider<in R, out T : Any>(
 val initializer: R.() -> T
) {
 operator fun provideDelegate(
  receiver: R,
  property: KProperty<*>
 ): CachedProperty<R, T> {
  if (property.annotations.any{ it is NoCache }) {
```

```
    throw IllegalStateException("${property.name} forbids
    caching")
   }
   return CachedProperty(initializer)
 }
}
class CachedProperty<in R, out T : Any>(val initializer: R.() ->
T) {
 private valcachedValues = HashMap<R, T>()
 operator fun getValue(receiver: R, property: KProperty<*>): T {
  return cachedValues.getOrPut(receiver) {
  receiver.initializer() }
 }
}
fun <R, T : Any> cached(initializer: R.() -> T) =
CachedPropertyProvider(initializer)
```

Now when we attempt to use caching delegate on a property with `@NoCache` annotation, the provider will fail with an error:

```
class Person(valfirstName: String, valfamilyName: String)

@NoCachevalPerson.fullName: String by cached {
 if (this != null) "$firstName $familyName" else ""
}

fun main() {
 val johnDoe = Person("John", "Doe")
 println(johnDoe.fullName) // Exception
}
```

Just like delegate accessors, the `provideDelegate()` may be implemented as either a member, or an extension function.

Delegate representation

Wrapping up our discussion of delegated properties, let’s talk about how delegates are represented and can be accessed at runtime.

At runtime, the delegate is stored in a separate field, while the property itself gets automatically generated accessors that invoked corresponding methods of the delegate instance. For example, the code:

```
class Person(val firstName: String, val familyName: String) {
 var age: Int by finalLateInit()
}
```

is effectively equivalent to the following with the exception that the delegate field `age$delegate` can't be used in Kotlin code explicitly:

```
class Person(val firstName: String, val familyName: String) {
 private val `age$delegate` = finalLateInit<Person, Int>()
 var age: Int
 get() = `age$delegate`.getValue(this, this::age)
 set(value) {
  `age$delegate`.setValue(this, this::age, value)
 }
}
```

The reflection API allows you to access the delegate value using the corresponding property object via its `getDelegate()` member. The signature varies depending on the number of receivers. For example:

```
import kotlin.reflect.jvm.isAccessible

class Person(val firstName: String, val familyName: String) {
 val fullName by lazy { "$firstName $familyName" }
}

fun main() {
 val person = Person("John", "Doe")
 // KProperty0: all receivers are bound
 println(
  person::fullName
  .apply { isAccessible = true }
  .getDelegate()!!::class.qualifiedName
   ) // kotlin.SynchronizedLazyImpl
   // KProperty1: single receiver
  println(
  Person::fullName
  .apply { isAccessible = true }
  .getDelegate(person)!!::class.qualifiedName
 ) // kotlin.SynchronizedLazyImpl
}
```

Please note the need to use `isAccessible = true` to access private field where delegate instance is stored.

What if our property is defined as an extension? In such a case, the delegate instance is shared among all the possible receivers and we can use `getExtensionDelegate()` to obtain it without specifying any particular receiver instance:

```
val Person.fullName: String by cached { "$firstName $familyName"
```

```
}
fun main() {
 println(
  Person::fullName
  .apply { isAccessible = true }
  .getExtensionDelegate()!!::class.qualifiedName
 ) // CachedProperty
}
```

Higher-order functions and DSLs

In this section, we will demonstrate how to design a domain-specific language using type-safe builders. This task will not require any new knowledge. Instead, it will just rely on what we have already learned about higher-order functions in Kotlin.

Fluent DSL with infix functions

Our first example will demonstrate how to use infix functions for creating fluent APIs. We will create a simple DSL for querying collection data, using SQL-inspired syntax.

In other word, we would like to be able to write code in the following manner (readers familiar with C# would probably recognize the similarity with LINQ):

```
val nums = listOf(2, 8, 9, 1, 3, 6, 5)
val query = from(nums) where { it > 3 } select { it*2 } orderBy {
it }
println(query.items.toList())
```

Basically, we want our query to consist of:

1. The `from` clause, which specifies an original collection,
2. Followed by an optional `where` clause, which specifies the filtering condition;
3. Followed by an optional `select` clause, which maps original data to output values;
4. When select is present, we may use an optional `orderBy` clause as well, which specifies an ordering key.

So how do we implement an API like this? First, let’s define some classes

representing the intermediate structures of a query. Since most of them represent a kind of data set, be it an original collection or a result of filtering, we will introduce a common interface with an ability to return the resulting sequence of items:

```
interface ResultSet<out T> {
 val items: Sequence<T>
}
```

Now we can define classes representing the query components:

```
class From<out T>(private val source: Iterable<T>) : ResultSet<T>
{
 override val items: Sequence<T>
 get() = source.asSequence()
}
class Where<out T>(
 private val from: ResultSet<T>,
 private val condition: (T) -> Boolean
) :ResultSet<T> {
 override val items: Sequence<T>
 get() = from.items.filter(condition)
}

class Select<out T, out U>(
 private val from: ResultSet<T>,
 private val output: (T) -> U
) :ResultSet<U> {
 override val items: Sequence<U>
 get() = from.items.map(output)
}

class OrderBy<out T, in K : Comparable<K>>(
 private val select: ResultSet<T>,
 private valorderKey: (T) -> K
) :ResultSet<T> {
 override val items: Sequence<T>
 get() = select.items.sortedBy(orderKey)
}
```

Now that we have our building blocks in place, we can define a set of infix functions to link them together, according to our DSL requirements:

```
// where may follow from
infix fun <T> From<T>.where(condition: (T) -> Boolean) =
Where(this, condition)

// select may follow either from or where
```

```
infix fun <T, U> From<T>.select(output: (T) -> U) =
Select(this, output)
infix fun <T, U> Where<T>.select(output: (T) -> U) =
Select(this, output)

// orderBy may follow select
infix fun <T, K : Comparable<K>> Select<*, T>.orderBy(
 orderKey: (T) -> K
) = OrderBy(this, orderKey)
```

The last piece is the from() function, which starts a query:

```
fun <T>from(source: Iterable<T>) = From(source)
```

Now the original example:

```
val nums = listOf(2, 8, 9, 1, 3, 6, 5)
val query = from(nums) where { it> 3 } select { it*2 } orderBy {
it }
println(query.items.toList())
```

Will compile and correctly print:

```
[10, 12, 16, 18]
```

Please note, that type-safety ensures the rejection of queries that do not conform to our intended syntax. For example, the following code will not compile, since only one where clause is allowed:

```
val query = from(nums) where { it> 3 } where { it < 10 }
```

If we, however, want to permit multiple where clauses, then we just need to add one more infix function:

```
infix fun <T> Where<T>.where(condition: (T) -> Boolean) =
Where(this, condition)
```

Now let’s look at a more complex example, with nested structures.

Using type-safe builders

A common case in designing a DSL is the representation of hierarchical structures, where some domain objects can be nested inside others. In Kotlin, you have a powerful solution that allows you to express such structures in a somewhat declarative manner, by combining the builder functions with extension lambdas. Let’s see how they can be implemented through an

example of a simple component layout DSL.

Our goal will be an API, which would allow us to describe program UI in the following manner:

```
fun main() {
 val form = dialog("Send a message") {
  borderLayout {
   south = panel {
    +button("Send")
    +button("Cancel")
   }
   center = panel {
    verticalBoxLayout {
     +filler(0, 10)
     +panel {
      horizontalBoxLayout {
       +filler(5, 0)
       +label("Message: ")
       +filler(10, 0)
       +textArea("")
       +filler(5, 0)
      }
     }
     +filler(0, 10)
    }
   }
  }
 }
 form.size = Dimension(300, 200)
 form.isVisible = true
}
```

Basically, we want our DSL to do the following:

- Describe a hierarchical structure of UI components;
- Support the standard layout managers like `BorderLayout` or `BoxLayout`;
- Provide helper functions to create and initialize common components like buttons, text fields, panels and windows.

Figure 11.2 displays the window produced from the code showcased above:

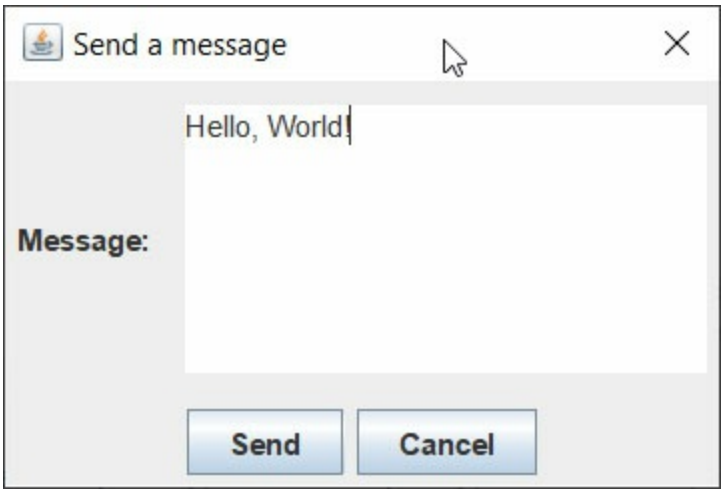

Figure 11.2: *A window generated by the layout DSL*

How do we implement a language of this kind? First, let's see what kind of objects are involved in UI description:

- Simple components like button or text field, which do not have a nested structure.
- Containers, like panels or windows: you may attach some layout to them or add nested components directly via the + operator.
- Layouts, which allow you to specify the child components of the corresponding container. The specifics depend on a particular layout: for example, a border layout binds the children to the predetermined areas (`NORTH`, `SOUTH`, etc.), while a box layout allows you to add components sequentially, placing them in a row or column.

Functions like `button()` are the simplest part, since they just wrap the component constructors without any extra processing:

```
fun label(text: String) = JLabel(text)
fun button(text: String) = JButton(text)
fun textArea(text: String) = JTextArea(text)
```

A more interesting case is the `panel()` function, which takes a lambda with the definitions of nested components. In order to maintain a container state, we will introduce the `ContainerBuilder` class, which allows you to add nested components and define layouts:

```
class ContainerBuilder(private val container: Container) {
 operator fun Component.unaryPlus() = apply {
 container.add(this) }
```

```
  fun borderLayout(body: BorderLayoutBuilder.() -> Unit) {
    BorderLayoutBuilder(container).body()
  }

  fun horizontalBoxLayout(body: BoxLayoutBuilder.() -> Unit) {
    BoxLayoutBuilder(container, BoxLayout.LINE_AXIS).body()
  }

  fun verticalBoxLayout(body: BoxLayoutBuilder.() -> Unit) {
    BoxLayoutBuilder(container, BoxLayout.PAGE_AXIS).body()
  }
}
```

Now we can define the `panel()` and the `dialog()` functions:

```
fun panel(body: ContainerBuilder.() -> Unit) = JPanel().apply {
  ContainerBuilder(this).body()
}
fun dialog(
  title: String,
  body: ContainerBuilder.() -> Unit
): JDialog = JDialog().apply {
  this.title = title
  pack()
  defaultCloseOperation = JDialog.DISPOSE_ON_CLOSE
  ContainerBuilder(contentPane).body()
}
```

You can see that these functions take a lambda, which serves as an extension on the `ContainerBuilder` class. This allows us to directly call the members of the `ContainerBuilder` inside a lambda, since this receiver is assumed implicitly. For example, the portion:

```
panel {
  horizontalBoxLayout {
    +filler(5, 0)
    …
  }
}
```

Really means:

```
panel {
  this.horizontalBoxLayout {
    // dispatch receiver of BoxLayoutBuilder is implicit
    filler(5, 0).unaryPlus()
    …
  }
```

```
}
```

What about layouts? We can define their builders in a similar way, remembering an API difference between various layouts. For example:

```
class BoxLayoutBuilder(private val container: Container,
direction: Int) {
 init {
  container.layout = BoxLayout(container, direction)
 }
 operator fun Component.unaryPlus() = apply {
 container.add(this) }
fun filler(width: Int, height: Int) =
 Box.createRigidArea(Dimension(width, height))
}
```

We've added `unaryPlus()` to `BoxLayoutBuilder` because, we want to add its children sequentially, like we do with a container like panel. In case of the `BorderLayoutBuilder`, we will need a set of properties like `north`, `south`, `west`, etc., which retain the value of the added component and adds it to the container, when changed. We can pack this logic into a variety of observable delegates:

```
fun constrained(
 container: Container,
 constraint: Any?
) = observable<Component?>(null) { _, _, value ->
 container.add(value, constraint)
}

class BorderLayoutBuilder(container: Container) {
 init {
  container.layout = BorderLayout()
 }

 var north by constrained(container, BorderLayout.NORTH)
 var south by constrained(container, BorderLayout.SOUTH)
 var west by constrained(container, BorderLayout.WEST)
 var east by constrained(container, BorderLayout.EAST)
 var center by constrained(container, BorderLayout.CENTER)
}
```

Many Kotlin DSLs are implemented in a similar manner. In the upcoming chapters, we will take a closer look at some languages targeted at the common tasks: test specification, description of UI in an Android application, request handling rules in a web application and type-safe generation of

HTML. For now, we have one more topic to discuss - how to control the scope of the builder functions.

@DslMarker

When using a hierarchical DSL, like the one that we discussed in the previous section, you may find that the members of the outer blocks are leaking into nested scopes. For example, using our layout DSL we could have written:

```
val myPanel = panel {
 borderLayout {
  borderLayout {

  }
 }
}
```

This is certainly unintended, because we did not want a layout-introducing function to be available on the layouts themselves. The code showcased above is still correct, and both the `borderLayout()` calls have the same receiver, in other words, the instance of the `ContainerBuilder` passed to the outermost lambda. The problem is that each of these receivers is available not only in its declaration scope, but also in all the nested scopes as well. If we would have made the receivers explicit, our code would look like:

```
val myPanel = panel {
 this@panel.borderLayout {
  this@panel.borderLayout {

  }
 }
}
```

Now, it is clear that both the receivers are the same.

So, even though the leakage of the implicit receivers into the nested scopes do not break the type safety, it certainly can be misleading and thus, lead to an error-prone code, especially if the code in question is a DSL that usually has a lot of nested extension lambdas. Hence, Kotlin 1.1 introduced `@DslMarker` annotation, which helps the DSL designers to restrict the visibility of implicit receivers.

`@DslMarker` is a meta-annotation that you use to annotate your own

annotation class, meant to serve as a marker of a particular DSL. Let's introduce `@LayoutDsl` for this purpose:

```
@DslMarker
annotation class LayoutDsl
```

Now we use `@LayoutDsl` to annotate classes that are used as receivers in DSL blocks. In our case, these are the `ContainerBuilder`, `BorderLayoutBuilder` and `BoxLayoutBuilder`:

```
@LayoutDsl
class ContainerBuilder(private val container: Container) {…}

@LayoutDsl
class BorderLayoutBuilder(container: Container) {…}

@LayoutDsl
class BoxLayoutBuilder(private val container: Container,
direction: Int)
{…}
```

If the classes in question have a common supertype, then you may annotate that supertype instead. DSL marker annotations automatically affect all subtypes.

Now that the compiler knows that these classes belong to the same DSL, it will not allow the use of corresponding receivers in the nested scopes. For example, our original fragment now produces a compilation error:

```
val myPanel = panel {
 borderLayout {
  borderLayout{ // Error: DSL scope violation

  }
 }
}
```

Please note, that the `@DslMarker` only forbids the leakage of implicit receivers. You can still use an explicit one if necessary:

```
val myPanel = panel {
 borderLayout {
  this@panel.borderLayout{ // Correct

  }
 }
}
```

Conclusion

This chapter has introduced us to the advanced features of the Kotlin language that help in designing internal, domain-specific languages, thus combining the simplicity of usage with type safety ensured by the Kotlin compiler. We have learned conventions that allow the developer to define overloaded operators, gone through standard implementations of delegated properties, and discussed how you can create your own. Finally, we have seen how functional programming, along with type-safe builders, can help in designing hierarchical DSLs.

In the next chapter, we will discuss the issue of Java/Kotlin interoperability. We will see how the Kotlin declarations can be used from Java code and vice versa, how to fine-tune your Kotlin-based API for Java clients and how the Kotlin tooling can help in converting a Java code into a Kotlin, automatically.

Questions

1. What are the operator overloading conventions available in Kotlin?
2. Describe standard delegate implementations.
3. What are the conventions used for property delegates? Give an example of a custom delegate implementation.
4. How can you access a delegate value at runtime?
5. Describe how to design a domain-specific language using higher-order functions.
6. Explain the meaning of the `@DslMarker` annotation.

CHAPTER 12

Java Interoperability

In this chapter, we will introduce you to various topics related to interoperability between Java and Kotlin code. This aspect plays a major role in mixed projects where the Java code has to coexist with Kotlin. Thanks to good JVM interoperability, you can easily add Kotlin to existing projects or gradually convert the Java code with little to no changes in its surroundings.

We'll look at how Kotlin and Java types map into each other, how Kotlin declarations are presented from the Java's point of view and vice versa and look at language features which can help you to customize Java/Kotlin interoperability.

Structure

- Using the Java code from Kotlin
- Using the Kotlin code from Java

Objectives

To learn how Kotlin declarations and types map into Java and how both languages can be mixed in a single codebase.

Using the Java code from Kotlin

Since Kotlin is designed with JVM as one of its primary targets, using the Java code in Kotlin is pretty straightforward. There are some issues mainly arising from the fact that some Kotlin features are not available in Java. For example, Java doesn't incorporate null safety into its type system while Kotlin always explicitly specifies whether they are nullable or not. Java types usually lack such information. In this section, we'll discuss how such issues can be resolved from both Java and Kotlin sides.

Java methods and fields

In most cases, using Java methods, Kotlin poses no concerns as they are exposed as ordinary Kotlin functions. Fields when they are not encapsulated are available like Kotlin properties with trivial accessor(s). Still it's worth bearing in mind some nuances stemming from language specifics.

`Unit` vs `void`

Kotlin has no void keyword representing the absence of the return value, so every void method in Java is visible as the `Unit` function in Kotlin. If you call such a function and use the call result somewhere (e.g. assign it to a variable), the compiler will generate a reference to the `Unit` object.

Operator conventions

Some Java methods such as `Map.get()` may satisfy Kotlin operator conventions. In Kotlin, you may use them in the operator form even though they do not have the operator keyword. For example, since the `Method` class from the Java Reflection API has the `invoke()` method, we can use it like a function:

```
val length = String::class.java.getDeclaredMethod("length")
println(length("abcde")) // 5
```

Infix calls, however, are not supported for Java methods.

Synthetic properties

Even though Java has no properties as such, using getters and setters is quite common. For this reason, the Kotlin compiler exposes getter/setter pairs as synthetic properties which you can access similar to ordinary Kotlin properties. The accessors must follow the following conventions:

- A getter must be a parameterless method and have a name starting with get.
- A setter must have a single argument and a name starting with set.

Suppose, for example, that you have the Java class:

```
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 void setName(String name) {
    this.name = name;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }
}
```

In Kotlin you can use its instances as if they had a pair of mutable properties such as `name` and `age`:

```
fun main() {
  val person = Person("John", 25)
  person.name = "Harry"
  person.age = 30
  println("${person.name}, ${person.age}") // Harry, 30
}
```

This convention also works when there is only getter method. In this case, a resulting property is immutable. When a Java class has a setter method, but no getter, no property is exposed since write-only properties are not currently supported in Kotlin.

As an alternative, a getter name may start with is. In this case, a synthetic property has the same name as getter. Suppose we extend the `Person` class above by adding a Boolean field with accessors:

```
public class Person {
  …

  private boolean isEmployed;
  public boolean isEmployed() {
```

```
    return isEmployed;
  }

  public void setEmployed(boolean employed) {
    isEmployed = employed;
  }
}
```

The Kotlin code can call these accessors using the `isEmployed` property.

IDE Tips: You can also use ordinary method calls instead of synthetic properties, but this is considered redundant. By default, the IntelliJ plugin warns about such calls suggesting you to replace them by the synthetic property access (see Figure 12.1):

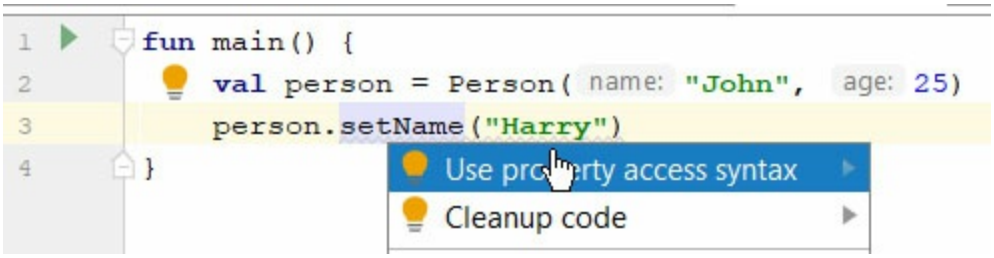

Figure 12.1: *Converting an explicit setter call to property assignment*

Note that the synthetic property syntax is only available for methods declared in a non-Kotlin code. You can't use the get/set methods defined in Kotlin source files for this purpose.

Platform types

Since Java doesn't distinguish between nullable and non-nullable types, the Kotlin compiler in general can't make any assumptions about nullability of objects coming from a Java code. Exposing them as nullable, however, is impractical because you'd have to deal with a lot of bogus nullability checks in the Kotlin code. For this reason, the Kotlin compiler relaxes null-safety when it comes to Java types and doesn't expose them as types with definite nullability. In Kotlin, objects originating from the Java code belong to special platform types which basically constitute type ranges between nullable and non-nullable versions. Null-safety guarantees provided by such types are basically the same as in Java. You can use their values in both nullable and non-nullable context, but such usage may fail at runtime with `NullPointerException`.

Consider the following Java class:

```
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 void setName(String name) { this.name = name; }
 public int getAge() { return age; }
 public void setAge(int age) { this.age = age; }
}
```

Let's try using it from a Kotlin code:

```
fun main() {
 val person = Person("John", 25)
 println(person.name.length) // 4
}
```

In the preceding code, `person.name` has a platform type because the compiler doesn't know if it can be nullable. Nevertheless, the code compiles but a nullability check is deferred to runtime when the program tries to access the `length` property. If we change it to the following code, the program will still compile, but fail at runtime:

```
fun main() {
 val person = Person(null, 25)
 println(person.name.length) // Exception
}
```

Note that platform types may not be written explicitly in a Kotlin source code. They are only constructed by a compiler. You can, however, see them in the IntelliJ IDEA plugin. For example, if you apply the **`Show expression type`** action (Ctrl+Shift+P/Cmd+Shift+P) to the `person.name` expression, you'll see that its type is String! (See Figure 12.2). This notation means that values of such type can pose as both values of `String?` and `String`:

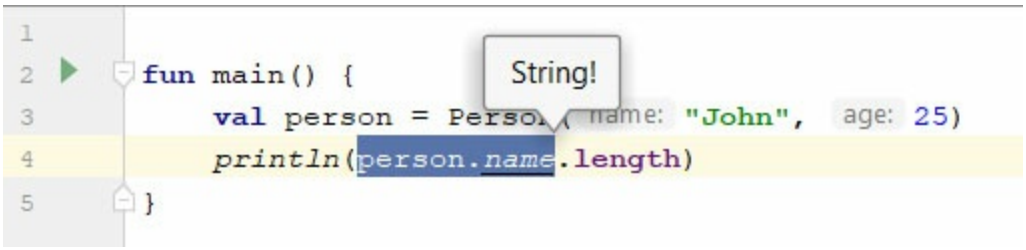

Figure 12.2: *Platform type representation in the IntelliJ IDEA*

If you assign an expression of a platform type to a variable or return it from a function without specifying an explicit type, it propagates. For example:

```
import java.math.BigInteger
// BigInteger! return type
fun Int.toBigInt() = BigInteger.valueOf(toLong())
val num = 123.toBigInt() // BigInteger! type
```

If you specify a type explicitly, you force the platform type into either nullable or non-nullable:

```
import java.math.BigInteger
// BigInteger (non-nullable) return type
fun Int.toBigInt(): BigInteger = BigInteger.valueOf(toLong())
val num = 123.toBigInt() // BigInteger (non-nullable) type
```

IDE Tips: The IntelliJ plugin can warn you about the implicit propagation of platform types and suggest either specifying type explicitly or adding not-null assertion !! (as shown in Figure 12.3):

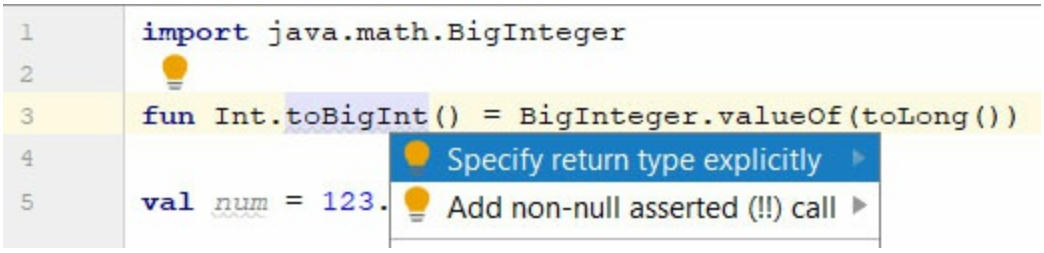

Figure 12.3: *Getting rid of the platform type propagation*

If we force a platform type into a non-nullable one, the compiler will generate an assertion. This ensures that the program will fail during the assignment rather than at some later moment when the assigned value is accessed.

Kotlin also uses platform types to represent Java collection types. The reason is similar to that of nullable types. Unlike Kotlin, Java doesn't distinguish between mutable or immutable collections. So, in Kotlin, each Java-

originating instance of standard collection type such as `List`, `Set` or `Map` looks like some range between mutable and mutable versions. In the IDE such types are represented by adding the (Mutable) prefix (for example, Figure 12.4).

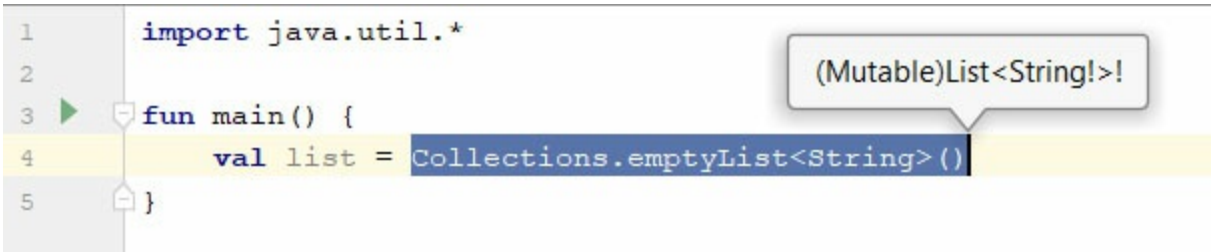

Figure 12.4: *Mutable platform types*

Nullability annotations

In the Java world, a common solution to a null-safety problem is using special type annotations. Modern development environments like IntelliJ IDEA can make use of such annotations reporting potential violations of nullability contracts. Some of them are supported by the Kotlin compiler as well. In this case, a respective type is exposed as either nullable or non-nullable (depending on the annotation used) and a platform type is not used. For example, if we annotate the Person class from our earlier example:

```
import org.jetbrains.annotations.NotNull;

public class Person {
 @NotNull private String name;
 private int age;

 public Person(@NotNull String name, int age) {
    this.name = name;
    this.age = age;
  }

  @NotNull public String getName() { return name; }
  public void setName(@NotNull String name) { this.name = name;
  }
  public int getAge() { return age; }
  public void setAge(int age) { this.age = age; }
}
```

Types in the Kotlin code will reflect these changes as shown in Figure 12.5:

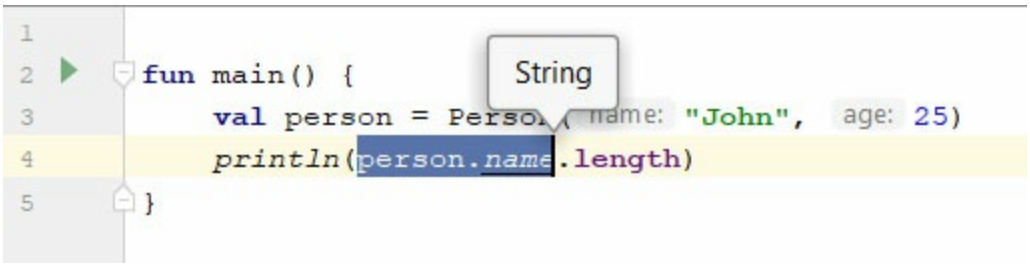

Figure 12.5: *Exposing Java type marked with the @NotNull annotation*

Some of the nullability annotations supported by the Kotlin compiler include (you can find a more comprehensive list in the Kotlin documentation at `kotlinlang.org`):

- JetBrains `@Nullable` and `@NotNull` (from `org.jetbrains.annotations` package)
- Multiple varieties of `@Nullable` and `@NonNull` annotations from Android SDK
- JSR-305 nullability annotations such as `@Nonnull` (from `javax.annotation` package)

IDE Tips: The JetBrains annotations library is not added to project dependencies automatically but can be easily configured when necessary. If the `@Nullable/@NotNull` annotation is not available, you can press Alt+Enter on the unresolved annotation reference and choose Add annotations to the classpath action as shown in Figure 12.6:

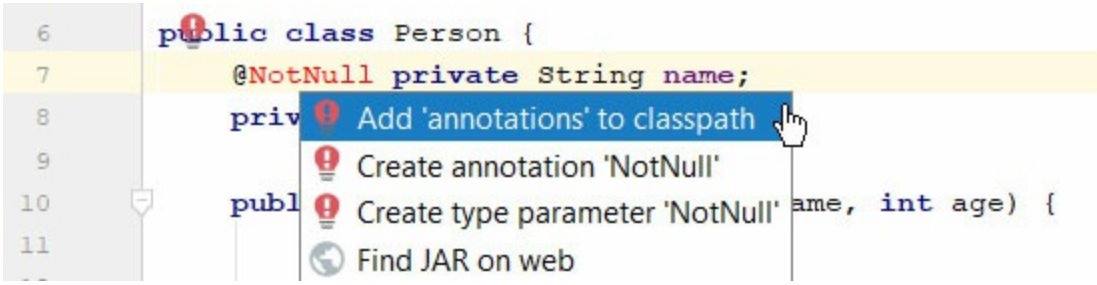

Figure 12.6: *Configuring JetBrains annotations library*

Note that since Java 8, you can also annotate type parameters of generic declarations provided the nullability annotation supports the `ElementType.TYPE_USE` target. For example, JetBrains `@Nullable/@NotNull` annotations support this target starting from version 15 so we can write the

following code:

```
public class Person {
 …
 @NotNull private Set<@NotNull Person> friends = new HashSet<>
 ();
 @NotNull public Set<@NotNull Person>getFriends() { return
 friends; }
}
```

In Kotlin, the return type of the `getFriends()` method will look like `(Mutable)Set<Person>`:

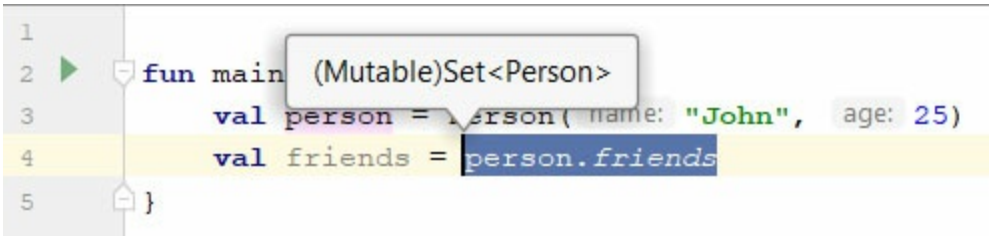

Figure 12.7: *Non-nullable type parameter*

When type parameters are not annotated, the Kotlin compiler has to use platform types for them, so the type of `person.friends` in the following Kotlin sample code will be `(Mutable)Set<Person!>` instead:

```
public class Person {
 …
 @NotNull private Set<Person> friends = new HashSet<>();
 @NotNull public Set<Person>getFriends() { return friends; }
}
```

Java/Kotlin type mapping

Some types have a similar meaning in Kotlin and Java. For example, Java primitive types like `int` and `boolean` correspond to built-in types of Kotlin (`Int` and `Boolean` respectively) while `java.util.List` corresponds to a platform type `(Mutable)List`. The Kotlin compiler is able to map Java types to their Kotlin counterparts when processing usage of Java declarations in the Kotlin code and vice versa when compiling the Kotlin code for JVM. In this section, we'll discuss the basic rules of Java/Kotlin type mapping.

First, Java primitive types as well as their boxed versions map to the corresponding basic types in Kotlin:

Java type	Kotlin type
byte/Byte	Byte
short/Short	Short
int/Integer	Int
long/Long	Long
char/Character	Char
float/Float	Float
double/Double	Double

This mapping also works in reverse. JVM values of basic Kotlin types are represented by either JVM primitive types or corresponding boxing classes depending on how that value is used. Value of `Int?`, say, would be represented by an instance of `java.lang.Integer` since null may not be stored as a value of Java's `int`.

Some non-primitive built-in classes from `java.lang` packages are mapped to the corresponding classes from the `kotlin` package (and vice versa). The names of classes are the same in both cases; the only exception being `Object`, which is mapped to Kotlin's `Any`:

- `Object`
- `Cloneable`
- `Comparable`
- `Enum`
- `Annotation`
- `CharSequence`
- `String`
- `Number`
- `Throwable`

Note that static members of the mapped Java classes (e.g. `Long.valueOf()`) are not accessible directly to companions of their Kotlin counterparts. To use them, you need to mention the qualified name of the corresponding Java class:

```
val n = java.lang.Long.bitCount(1234)
```

Standard collection types in Kotlin (both mutable and immutable) are mapped to the corresponding collection types from the `java.util` package. Reverse mapping, as discussed earlier, produces platform types because standard Java collections use the same API for both mutable and immutable implementations. The mapped types are as follows:

- Iterable/Iterator/ListIterator
- Collection
- Set
- List
- Map/Map.Entry

Mapping between generic types involve some less trivial transformation due to differences in their syntax:

- `extends` wildcards in Java correspond to covariant projections in Kotlin. For example, `TreeNode<? extends Person>` maps to `TreeNode<out Person>`.
- `super` wildcards map to contravariant projections: `TreeNode<? super Person> vs. TreeNode<in Person>`.
- Raw types in Java are represented by types with star projections: `TreeNode` becomes `TreeNode<*>`.

Java arrays of primitive types (like `int[]`) are mapped to the corresponding specialized array classes (e.g. `IntArray`) to avoid boxing/unboxing operations. Any other array is represented as an instance of a special platform type `Array<(out) T>` (which may also be a nullable platform: e.g. `Array<(out) String>!`) which combines `Array<T>` and `Array<out T>`. This, in particular, allows you to pass an array of subtypes to Java methods expecting an array of supertypes. For example, the following code passes an array of `String` as a value of the `Object[]` parameter:

```
import java.util.*
fun main() {
 val strings = arrayOf("a", "b", "c")
 println(Arrays.deepToString(strings))
}
```

Such behavior is consistent with Java semantics where array types are

covariant. Kotlin arrays are invariant so this trick doesn't work with Kotlin methods unless you restrict the `Array` type to its out-projection as in `Array<out Any>`.

Single abstract method interfaces

If you have a Java interface with a single abstract method (SAM interface for short), it essentially behaves like a Kotlin functional type.This is in fact similar to Java 8+ which supports automatic conversion of lambdas into an appropriate SAM type instance. Kotlin gives you an ability to use lambdas in context where Java SAM interface is expected. This is called a SAM conversion. For example, let's look at the JDK `ExecutorService` class whose API allows you to register some tasks for asynchronous computations. Its `execute()` method takes a `Runnable` object:

```
public interface Runnable {
 public void run();
}
```

Since `Runnable` has is qualified as a SAM interface in Kotlin, this allows the Kotlin code to call the `execute()` methods by simply passing a lambda:

```
import java.util.concurrent.ScheduledThreadPoolExecutor
fun main() {
 val executor = ScheduledThreadPoolExecutor(5)
 executor.execute {
  println("Working on asynchronous task…")
 }
 executor.shutdown()
}
```

Instead of much more verbose code:

```
import java.util.concurrent.ScheduledThreadPoolExecutor
fun main() {
 val executor = ScheduledThreadPoolExecutor(5)
 executor.execute(object : Runnable {
  override fun run() {
   println("Working on asynchronous task…")
  }
 })
 executor.shutdown()
}
```

IDE Tips: The IntelliJ plugin can warn you about unnecessary object expressions like the one above and automatically convert them to implicit SAM conversions (as shown in Figure 12.8).

Figure 12.8: *Converting an object expression to lambda*

Sometimes a compiler doesn't have enough context information to choose a proper conversion. Say, Java `ExecutorService` has a set of the `submit()` methods which take an object representing some computation to execute it in future. Computation may be an instance of either the Runnable or Callable interface which looks like the following:

```
public interface Callable<V> {
 V call() throws Exception;
}
```

Both `Runnable` and `Callable` are SAM interfaces, but if we pass a lambda to one of the `submit()` methods in a Kotlin code, the compiler will choose a `Runnable` version because it has the most specific signature:

```
import java.util.concurrent.ScheduledThreadPoolExecutor
fun main() {
 val executor = ScheduledThreadPoolExecutor(5)
 // implicitly converted to Runnable
 val future = executor.submit { 1 + 2 }
 println(future.get()) // null
 executor.shutdown()
}
```

What if we want to pass a `Callable` instance instead? In this case, we have to make the conversion more explicit by specifying a target type:

```
import java.util.concurrent.Callable
import java.util.concurrent.ScheduledThreadPoolExecutor
fun main() {
 val executor = ScheduledThreadPoolExecutor(5)
 val future = executor.submit(Callable { 1 + 2 })
 println(future.get()) // 3
```

```
  executor.shutdown()
}
```

Such an expression is called a SAM constructor.

Note that SAM conversions only work for interfaces and not for classes even if they have a single abstract method. They also do not work with Kotlin interfaces; unlike Java, Kotlin has proper functional types so implicit conversion is virtually unnecessary.

Using Java to Kotlin converter

The IntelliJ plugin includes an automatic tool which can convert the Java source file to an equivalent Kotlin code. Together with Java/Kotlin interoperability, this feature allows you to gradually migrate to the existing Java codebase.

To convert a file, you just need to press Ctrl+Alt+Shift+K or choose the **`Convert Java File to Kotlin File`** action from the **`Code`** menu. The IDE will then process your file, convert it to Kotlin and update external usages if necessary.

You can also select one or several files in the **`Project View`** panel and apply the same shortcut to convert them in a single batch.

The automatic converter aims at producing idiomatic Kotlin code, but it doesn't always produce an ideal result. Nevertheless this tool can be used as a good starting point when migrating existing Java codebase to Kotlin.

Using the Kotlin code from Java

One of the Kotlin design guidelines is a smooth interoperability with an existing Java codebase. In most cases, the Kotlin code can be easily used from Java without much concern. Kotlin, however, possesses a number of features which do not have direct counterparts in Java. In this section, we'll talk about these nuances as well as discuss how you can fine-tune the Kotlin code exposition from Java's point of view.

Accessing properties

Since neither Java nor JVM have a concept of property, you can't access Kotlin properties directly from the Java code. In the compiled JVM bytecode,

however, each property is represented by accessor methods which are available to Java clients, just like any ordinary methods. Accessor signatures are derived from the property definition according to the following rules:

- A getter is a parameterless method with a return type corresponding to the original property type; its name is computed by uppercasing the first letter of the property name and prefixing it with get.
- A setter is a void method which takes a single parameter corresponding to a new value; its name is similar to getter's, although withget being replaced by set.

For example, consider the following Kotlin class:

```
class Person(var name: String, val age: Int)
```

This will look like the following (from Java's point of view):

```
public class Person {
 @NotNull
 public String getName() {…}
 public void setName(@NotNull String value) {…}
 public int getAge() {…}
}
```

So, a Java client code can access its properties by calling the accessor methods:

```
public class Main {
 public static void main(String[] args) {
  Person person = new Person("John", 25);
  System.out.println(person.getAge()); // 25
  person.setName("Harry");
  System.out.println(person.getName()); // Harry
 }
}
```

When a property name starts with is, the Kotlin compiler uses another naming scheme as follows:

- The getter has the same name as its property.
- The setter name is computed by replacing the is prefix with set.

For example, suppose that we add the `isEmployed` property to our Person class:

```
class Person(var name: String, val age: Int, var isEmployed:
Boolean)
```

The Java code accessing the new property will now look like this:

```
public class Main {
 public static void main(String[] args) {
  Person person = new Person("John", 25, false);
  person.setEmployed(true);
  System.out.println(person.isEmployed()); // true
 }
}
```

Note that is convention is purely name-based; it has nothing to do with the `Boolean` type (although, it's strongly recommend to use is names for Boolean properties only to avoid misunderstanding).

If the Kotlin property requires a backing field, the compiler will generate it alongside the accessor method(s). By default, however, this field is private and can't be accessed directly outside of the getter/setter code. In some cases, you may need to expose that property to Java clients. This can be achieved by annotating the property with `@JvmField`. For example, if we modify our Person class by annotating its constructor parameters:

```
class Person(@JvmField var name: String, @JvmFieldval age: Int)
```

We can access the generated fields from the Java source code:

```
public class Main {
 public static void main(String[] args) {
  Person person = new Person("John", 25);
  System.out.println(person.age); // 25
  person.name = "Harry";
  System.out.println(person.name); // Harry
 }
}
```

In this case, the accessor methods are not generated and the backing field has the same visibility level as the property itself. Note that `@JvmField` can't be used if the property has non-trivial accessors:

```
class Person(valfirstName: String, valfamilyName: String) {
 // Error: property has a custom getter
 @JvmFieldvalfullNameget() = "$firstName $familyName"
}
```

`@JvmField` is also not applicable to open or abstract properties since their

overrides may in general have custom accessors:

```
open class Person(val firstName: String, val familyName: String)
{
 // Error: property is open
 @JvmField open val description: String
 get() = “$firstName $familyName”
}
```

When applied to a property of some named object, `@JvmField` behaves a little differently generating a static field instead of an instance one. Consider, for example, the Kotlin object:

```
object Application {
 @JvmFieldval name = "My Application"
}
```

The Java code can access the `name` property by referring to the `Application.name` field directly:

```
public class Main {
 public static void main(String[] args) {
  System.out.println(Application.name);
 }
}
```

The same also goes for properties with the `const` modifier:

```
object Application {
 const val name = "My Application"
}
```

Another way to expose the backing field is to use a `lateinit` property:

```
class Person(val firstName: String, val familyName: String) {
 lateinit var fullName: String

 fun init() {
  fullName = "$firstName $familyName"
 }
}
```

In this case, both accessors and the backing field have the same visibility as the property itself:

```
public class Main {
 public static void main(String[] args) {
  Person person = new Person("John", "Doe");
```

```
    person.init();
    // direct field access
    System.out.println(person.fullName); // John Doe
    // accessor call
    System.out.println(person.getFullName()); // John Doe
  }
}
```

In objects, `lateinit` generates a static field similar to the `@JvmField` annotation. Its accessors, however, remain instance methods. Let's, for example, define the singleton:

```
object Application {
  lateinit var name: String
}
```

The Java code below shows the difference between accessing the lateinit property's field and its accessor methods:

```
public class Main {
  public static void main(String[] args) {
    // Accessor call (non-static)
    Application.INSTANCE.setName("Application1");
    // Direct property access (static)
    Application.stdin = "Application2"
  }
}
```

Note that `@JvmField` can't be used for `lateinit` properties.

File facades and top-level declarations

In Kotlin, you can often make use of top-level declarations which are placed directly in a package rather than inside some other declaration. The Java and JVM platforms in general, these methods must always belong to a particular class. To satisfy this requirement, the Kotlin compiler puts top-level functions and properties into an automatically generated class which is called a file facade. By default, the facade name is based on the name of the source file with extra Kt suffix. For example, the file:

```
// util.kt
class Person(val firstName: String, val familyName: String)

val Person.fullName
get() = "$firstName $familyName"
```

```
fun readPerson(): Person? {
 val fullName = readLine() ?: return null
 val p = fullName.indexOf(' ')
 return if (p >= 0) {
  Person(fullName.substring(0, p), fullName.substring(p + 1))
 } else {
 Person(fullName, "")
 }
}
```

will produce the following facade class:

```
public class UtilKt {
 @NotNull
 public static String getFullName(@NotNull Person person) {…}
 @Nullable
 public static Person readPerson() {…}
}
```

Note that the facade class doesn’t contain classes since they are allowed at top-level in both JVM and Java.

Since generated methods are static, you don’t need to instantiate the facade class when using it from the Java code:

```
public class Main {
 public static void main(String[] args) {
  Person person = UtilKt.readPerson();
  if (person == null) return;
  System.out.println(UtilKt.getFullName(person));
 }
}
```

The Kotlin compiler allows you to tune some aspects of generated facade. First, you can change its name by adding the file-level `@JvmName` annotation:

```
@file:JvmName("MyUtils")
class Person(val firstName: String, val familyName: String)
val Person.fullName
 get() = "$firstName $familyName"
```

Now, its Java clients will need to use the specified MyUtils name:

```
public class Main {
 public static void main(String[] args) {
  Person person = new Person("John", "Doe");
  System.out.println(MyUtils.getFullName(person));
 }
}
```

Another useful ability is to merge top-level declarations from multiple files into a single class. To do this, you need to annotate files of interest with `@JvmMultifileClass` and specify the target class name with `@JvmName`. In this case, the Kotlin compiler will automatically combine files with the same facade class name. For example, suppose that all declarations from our example are written in separate files:

```
// Person.kt
class Person(val firstName: String, val familyName: String)

// utils1.kt
@file:JvmMultifileClass
@file:JvmName("MyUtils")

val Person.fullName
get() = "$firstName $familyName"

// utils2.kt
@file:JvmMultifileClass
@file:JvmName("MyUtils")

fun readPerson(): Person? {
 val fullName = readLine() ?: return null
 val p = fullName.indexOf(' ')
 return if (p >= 0) {
  Person(fullName.substring(0, p), fullName.substring(p + 1))
 } else {
  Person(fullName, "")
 }
}
```

Thanks to `@JvmMultifile` and `@JvmName`, we can still access both declarations as members of the `MyUtils` class:

```
public class Main {
 public static void main(String[] args) {
  Person person = MyUtils.readPerson();
  if (person == null) return;
  System.out.println(MyUtils.getFullName(person));
 }
}
```

Note that facade classes are not available to the Kotlin code and they are only usable by other JVM clients.

Objects and static members

On JVM, Kotlin object declarations are compiled into ordinary classes with the static `INSTANCE` field. For example, if we have the following Kotlin declaration:

```
object Application {
 val name = “My Application”
 fun exit() { }
}
```

the Java code can access its members using the `Application.INSTANCE` field:

```
public class Main {
 public static void main(String[] args) {
  System.out.println(Application.INSTANCE.getName());
  Application.INSTANCE.exit();
 }
}
```

We’ve already seen that using `@JvmField` on an object property turns it into a static field from Java’s point of view. Sometimes, it can be useful to make object functions or property accessors available as static methods. To do this, you can use the `@JvmStatic` annotation:

```
importjava.io.InputStream
object Application {
 @JvmStatic var stdin: InputStream = System.`in`
 @JvmStatic fun exit() { }
}
```

In the Java code, such functions and properties can be invoked without mentioning a particular instance:

```
import java.io.ByteArrayInputStream;
public class Main {
 public static void main(String[] args) {
  Application.setStdin(new
  ByteArrayInputStream("hello".getBytes()));
  Application.exit();
 }
}
```

Changing the exposed declaration name

We’ve already seen how `@JvmName` can be used to specify the name of the facade class for top-level declarations. In fact, this annotation is applicable

not only to files, but also to functions and property accessors. It allows you to change the name of the corresponding JVM methods.

The primary use of this feature is the ability to resolve signature clashes between declarations which are valid in Kotlin but forbidden in Java. Consider the following Kotlin code:

```
class Person(val firstName: String, val familyName: String)
val Person.fullName
   get() = "$firstName $familyName" // Error
fun getFullName(person: Person): String { // Error
 return "${person.familyName}, ${person.firstName}"
}
```

This code will produce a compilation error even though the Kotlin client can easily distinguish function and property on JVM. Both declarations will produce a method with the same signature, thus leading to ambiguity:

```
@NotNull
public static String getFullName(@NotNull Person person) {…}
```

Using `@JvmName`, you can change the conflicting name and fix the problem:

```
@JvmName(“getFullNameFamilyFirst”)
fun getFullName(person: Person): String { // Error
 return "${person.familyName}, ${person.firstName}"
}
```

Now, the Java client will be able to call this function using the `getFullNameFamilyFirst` name, while the Kotlin code will use the original `getFullName`.

We can similarly specify the JVM name for properties by annotating its particular accessor(s):

```
val Person.fullName
@JvmName("getFullNameFamilyLast")
 get() = "$firstName $familyName"
```

Or to the property itself (with appropriate use-site target):

```
@get:JvmName("getFullNameFamilyLast")
val Person.fullName
 get() = "$firstName $familyName"
```

`@JvmName`, in particular, allows you to circumvent the standard naming scheme used for property accessors:

```
class Person(@set:JvmName("changeName") var name: String, val
age: Int)
```

When seen from the Java code, the Person class will now have the `changeName()` method instead of `setName()`:

```
public class Main {
 public static void main(String[] args) {
  Person person = new Person("John", 25);
  person.changeName("Harry");
  System.out.println(person.getName());
 }
}
```

`@JvmName` is also useful when Kotlin function's name coincides with the Java keyword which makes it unusable from the Java source code. For example:

```
class Person(val firstName: String, val familyName: String) {
 @JvmName("visit")
 fun goto(person: Person) {
  println("$this is visiting $person")
 }
 override fun toString() = "$firstName $familyName"
}
```

The `goto()` function is not callable in Java since `goto` is a reserved keyword. Providing custom JVM name fixes the problem.

Generating overloads

When a Kotlin function has parameters with a default value, the number of arguments in its call may vary since some of them may be skipped:

```
// util.kt
fun restrictToRange(
 what: Int,
 from: Int = Int.MIN_VALUE,
 to: Int = Int.MAX_VALUE
): Int {
 return Math.max(from, Math.min(to, what))
}
```

```
fun main() {
 println(restrictToRange(100, 1, 10)) // 10
 println(restrictToRange(100, 1)) // 100
 println(restrictToRange(100)) // 100
}
```

Java, however, has no concept of default values, so the preceding function will look like this:

```
public int restrictToRange(int what, int from, int to) {…}
```

As a result, any Java client would be forced to explicitly pass all arguments:

```
public class Main {
 public static void main(String[] args) {
  System.out.println(UtilKt.restrictToRange(100, 1, 10));
  System.out.println(UtilKt.restrictToRange(100, 1)); // Error
  System.out.println(UtilKt.restrictToRange(100)); // Error
 }
}
```

Kotlin gives you a solution with the `@JvmOverloads` annotation:

```
@JvmOverloads
fun restrictToRange(
 what: Int,
 from: Int = Int.MIN_VALUE,
 to: Int = Int.MAX_VALUE
): Int {
 return Math.max(from, Math.min(to, what))
}
```

The effect of `@JvmOverloads` is to generate additional overloads for an original Kotlin function:

- The first one has all parameters of the original except the last parameter with a default value.
- The second one has all parameters of the original except the last and the second-to-last parameters with a default value and so on.
- The last overloaded version has only parameters without default values.

For example, the `restrictToRange()` function now has three overloads from Java's point of view:

```
public int restrictToRange(int what, int from, int to) {…}
public int restrictToRange(int what, int from) {…}
public int restrictToRange(int what) {…}
```

Additional overloads will call an original function providing explicit values for omitted arguments. Now, our original Java usages become valid as all three overloads are correctly resolved:

```
public class Main {
 public static void main(String[] args) {
  System.out.println(UtilKt.restrictToRange(100, 1, 10)); // 10
  System.out.println(UtilKt.restrictToRange(100, 1)); // 100
  System.out.println(UtilKt.restrictToRange(100)); // 100
 }
}
```

Note that although overloaded versions, produced by the `@JvmOverloads` annotation, are added to compiled binaries, they are not available in the Kotlin code. These overloads are meant to be used for Java interoperability only.

Declaring exceptions

In *Chapter 3, Defining Functions*, we mentioned that Kotlin doesn't distinguish between checked and unchecked exceptions; your functions and properties may throw arbitrary exceptions without any extra code. Java, on the other hand, requires to explicitly list checked exceptions which are not caught in method body. This may lead to problems if the Java code wants to handle a checked exception which may be thrown by calling a Kotlin declaration. For example, suppose that we have a Kotlin function:

```
// util.kt
fun loadData() = File("data.txt").readLines()
```

And use from the Java side:

```
public class Main {
 public static void main(String[] args) {
  for (String line :UtilKt.loadData()) {
   System.out.println(line);
  }
 }
}
```

If data.txt can't be read, `loadData()` throws `IOException` which remains unhandled thus leading to silent failure of the `main()` method. If we try to add an exception handler to `main()`, we may face another problem:

```
import java.io.IOException;
public class Main {
 public static void main(String[] args) {
  try {
   for (String line :UtilKt.loadData()) {
```

```
      System.out.println(line);
    }
   } catch (IOException e) { // Error
    System.out.println("Can't load data");
   }
  }
}
```

Compilation fails because Java forbids handling of checked exceptions which are not declared inside the corresponding try block. The problem is that from Java's point of view our `loadData()` function looks like this:

```
@notNull
public List<String>loadData() {…}
```

And so it doesn't provide any information about possibly thrown exceptions. The solution is to use the special `@Throws` annotation where you can specify exception classes:

```
// util.kt
@Throws(IOException::class)
fun loadData() = File("data.txt").readLines()
```

Now, we can properly handle its call within Java's `try-catch block`. Calling it outside of the exception handler or any method with an explicit throws `IOException` clause will lead to compilation error as expected:

```
public class Main {
 public static void main(String[] args) {
  // Error: Unhandled IOException
  for (String line :UtilKt.loadData()) {
   System.out.println(line);
  }
 }
}
```

Bear in mind that the Kotlin compiler doesn't validate consistency of `@Throws` annotations between base and overriding members. For example, we can write:

```
import java.io.File
import java.io.IOException
abstract class Loader {
 abstract fun loadData(): List<String>
}
class FileLoader(val path: String) : Loader() {
 @Throws(IOException::class)
```

```
 override fun loadData() = File(path).readLines()
}
```

Such class hierarchy can't be declared in the Java source code since language specification forbids adding extra checked exceptions in overriding methods.

Inline functions

Since Java has no notion of inline functions, Kotlin functions marked with inline modifiers are exposed as ordinary methods. You can call them in the Java code, but their bodies are not inlined in this case.

A special case is a generic inline function with reified type parameter(s). As of now, type reification can't be implemented without inlining so calling such functions from the Java code is not possible. For example, the following `cast()` function is not usable for Java clients:

```
inline fun <reified T : Any>Any.cast(): T? = this as? T
```

It will be exposed as a private member of the facade class, thus forbidding any external access:

```
public class Main {
 public static void main(String[] args) {
  UtilKt.<Integer>cast(""); // Error: cast is private
 }
}
```

Type aliases

Kotlin type aliases can't be used in the Java code. Any declaration referring to type aliases will use its underlying type when seen from the Java. For example, from JVM's view, the following definitions would produce a `Person` class with `Name` alias replaced by `String`:

```
typealias Name = String
class Person(val firstName: Name, val familyName: Name)
```

This can demonstrated by the Java code which constructs and uses instances of the `Person` class:

```
public class Main {
 public static void main(String[] args) {
  Person person = new Person("John", "Doe");
```

```
    System.out.println(person.getFamilyName()); // Doe
  }
}
```

Conclusion

This chapter introduced us to how to mix Kotlin and Java code within a common codebase. We looked at how Kotlin and Java declarations are exposed to each other, which common problems can arise when you attempt to use Java declarations in the Kotlin code and vice versa and addressed their basic solutions as well as the means to tune language interoperability on the JVM platform.

In the next chapter, we will focus on the concurrent applications. We'll see how Java concurrent primitives can be used in Kotlin and discuss various aspects of coroutines, a powerful language feature which allows you to program suspendable computations.

Questions

1. What is a synthetic property? What are the rules regarding the use of Java accessor methods in Kotlin?
2. What is a platform type? What kinds of platform types are supported in Kotlin?
3. How nullability annotations in the Java code affect Kotlin types?
4. Describe how Kotlin types are mapped to Java and vice versa.
5. Explain how SAM conversions and constructors work in Kotlin.
6. How Kotlin properties can be accessed from the Java code?
7. In which cases, backing fields of Kotlin properties are available to the Java code?
8. What is a file facade? Describe how Kotlin top-level functions and properties can be used in Java.
9. How do you merge multiple Kotlin files into a single facade class?
10. Describe usages of `@JvmName` annotation.
11. Describe how instances of Kotlin object declarations are exposed to Java code.

12. How do you make object members available as static methods in Java?
13. What is an effect of using `@JvmOverloads`?
14. How would you declare possible checked exceptions for a Kotlin function?

CHAPTER 13

Concurrency

In this chapter, we will focus on a major topic of writing concurrent code. Our main goal will be to get an understanding of coroutines; one of the distinguishing features of Kotlin, first introduced in version 1.1 and achieving release status in Kotlin 1.3.

We'll start with the discussion of basic ideas underlying Kotlin coroutines such as suspending functions and structured complexity and then gradually move to more advanced issues of concurrent control-flow, how the coroutine state changes throughout its lifecycle, how cancellation and exception work, and how concurrent tasks get assigned to threads.

We will also cover techniques such as channels and actors which allow your code to implement communication between concurrent tasks and to share some mutable data in a thread-safe manner.

In conclusion, we'll also discuss some utilities simplifying the usage of the Java concurrency API in the Kotlin code, creating threads, and using synchronization and locks.

Structure

- Coroutines
- Concurrent communication
- Using Java concurrency

Objective

To learn using concurrency primitives provided by the Kotlin coroutines library for building scalable and responsive code.

Coroutines

Kotlin programs can easily use Java concurrent primitives to achieve thread safety. Using them, however, still poses a certain problem because most concurrent operations are blocking; in other words, a thread which uses operations such as `Thread.sleep()`, `Thread.join()` or `Object.wait()` will be blocked until its execution is completed. Blocking and resuming thread execution requires computationally intensive context switching at the system level which may negatively impact program performance. On top of that, each thread consumes a considerable amount of system resources so maintaining a large number of concurrent threads may be impractical or even not possible at all.

A more efficient approach involves asynchronous programming. We can supply a lambda which is to be called back when the requested operation is completed. The original thread in the meantime can go on with some useful work (like processing client requests or handling UI events) instead of just waiting in a blocked state. The major problem of such an approach is a drastic increase of code complexity as we can't use an ordinary imperative control-flow.

In Kotlin, you can have the best of both worlds; thanks to a powerful mechanism of coroutines which allows you to write code in a familiar imperative style and yet have it automatically transformed into an efficient asynchronous computation by compiler. This mechanism is based on the concept of suspending functions which are able to preserve their context and may be suspended and resumed at certain points of their execution.

It's worth noting that most of the coroutines power is provided by a separate library which must be explicitly configured in your project. The version used in the book is given by the following Maven coordinates: `org.jetbrains.kotlinx:kotlinx-coroutines-core:1.2.2`.

IDE Tips: If you're using IntelliJ IDEA without relying on any particular build system like Maven or Gradle, you can add the coroutines library by following the given steps:

1. Press F4 on the root node in the **`Project View`** panel or right click on it and choose **`Open Module Settings`**.
2. Click on the **`Libraries`** item on the left, then click on the + button on the top toolbar and choose the **`From Maven…`** option.

3. Type the Maven coordinates of the library (e.g. **`org.jetbrains.kotlinx:kotlinx-coroutines-core:1.2.2`**) and click OK (as shown in Figure 13.1).
4. The IDE will then download the library with the necessary dependencies and suggest adding it to modules of your project. Confirm it by clicking on OK.

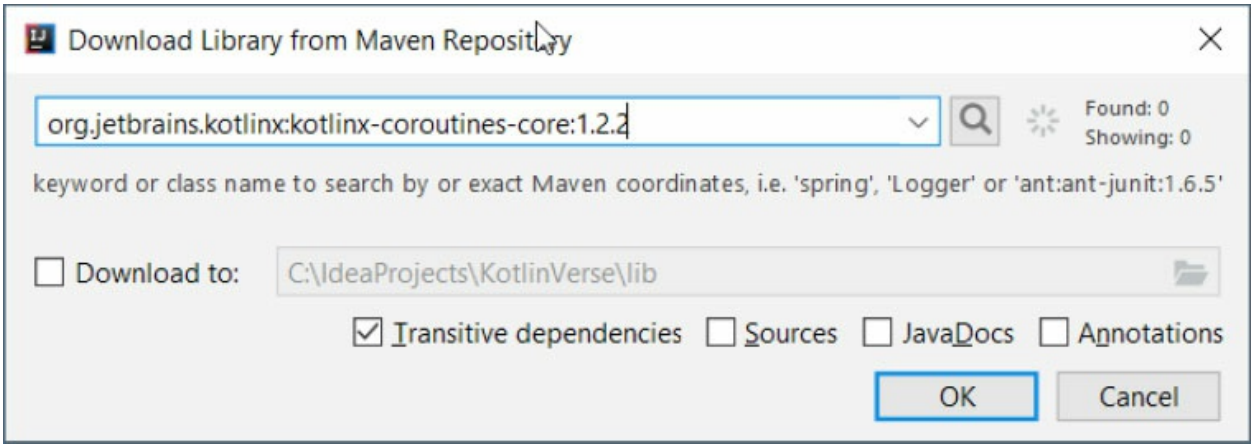

Figure 13.1: *Downloading Kotlin coroutines library*

In the following sections, we'll talk about the basic concepts introduced by the coroutines library and see how they can be used for the purpose of concurrent programming.

Coroutines and suspending functions

The basic language primitive underlying the entire coroutines library is a suspending function. This is generalization of an ordinary function which has the ability to suspend its execution on certain points in its body retaining all the necessary context and then resume on demand. In Kotlin, such functions are marked by the suspend modifier:

```
suspend fun foo() {
 println("Task started")
 delay(100)
 println("Task finished")
}
```

The `delay()` function we've used is a suspending function defined in the coroutines library. Its purpose is similar to the `Thread.sleep()`; however, instead of blocking the current thread, it suspends the calling function leaving

the thread free for execution of other tasks (such as switching to the other suspending function).

Suspending functions may call both suspending and ordinary functions. In the former case, such a call becomes a suspension point where the caller execution may be temporarily stopped and resumed later, while the latter proceeds just like a normal function call which returns after the invoked function has finished. Kotlin, however, forbids calling suspending functions from an ordinary one:

```
fun foo() {
 println("Task started")
 delay(100) // Error: delay is a suspend function
 println("Task finished")
}
```

IDE Tips: When using the IntelliJ plugin, you can easily distinguish the suspending call by a special icon on the left of the corresponding line as shown in Figure 13.1:

Figure 13.2: *Suspending calls in IDE*

If only suspending functions are allowed to make suspending calls, how do we invoke a suspending function at all? The most obvious way to is to mark the `main()` function itself as suspend:

```
import kotlinx.coroutines.delay
suspend fun main() {
 println("Task started")
 delay(100)
 println("Task finished")
}
```

When you run the preceding code, it prints the following with a 100 ms delay in between as expected:

```
Task started
Task finished
```

In more realistic cases, however, when we need some control over the behavior of our concurrent code, suspending functions are executed in a specific scope which defines a set of tasks with a shared lifecycle and context. Various functions which are used to launch coroutines are commonly known as coroutine builders, which serve as extensions on `CoroutineScope` instances. One of its basic implementations is given by the `GlobalScope` object; it allows us to create standalone coroutines which can further spawn their own nested tasks. Now, we'll take a look at three common coroutine builders: `launch()`, `async()`, and `runBlocking()`.

Coroutine builders

The `launch()` function starts a coroutine and returns a `Job` object which you can use to track its state and cancel the underlying task when needed. The function takes a suspending lambda of type `CoroutineScope.() -> Unit` which comprises the body of a new coroutine. Let's consider a simple example:

```
import kotlinx.coroutines.*
import java.lang.System.*

fun main() {
 val time = currentTimeMillis()
 GlobalScope.launch {
  delay(100)
  println("Task 1 finished in ${currentTimeMillis() - time} ms")
 }
 GlobalScope.launch {
  delay(100)
  println("Task 2 finished in ${currentTimeMillis() - time} ms")
 }
 Thread.sleep(200)
}
```

If you run the preceding code, you'll see something like this:

```
Task 2 finished in 176 ms
Task 1 finished in 176 ms
```

A thing worth noting is that both tasks complete at virtually the same time relative to the program start which means that they are really executed in parallel. A particular order is not guaranteed, so any of the two tasks may become the first depending on the circumstances. The coroutines library also includes means for enforcing execution ordering when that's necessary. We'll discuss them in the upcoming sections devoted to concurrent communication.

The `main()` function itself uses `Thread.sleep()` to temporarily block the main thread execution. This should give coroutine threads enough time to complete since by default they are run in the daemon mode and would be shut down early once the `main()` thread terminates.

Bear in mind that using thread-blocking functions like `sleep()` is possible inside the suspending function as well, but you're strongly discouraged from doing this because such code defeats the entire purpose of coroutines. For this reason, we've used suspending `delay()` inside our concurrent tasks.

> **IDE Tips: The IntelliJ plugin warns you about potentially blocking function calls like `Thread.sleep()` or `Thread.join()` inside the coroutine code.**

Note that coroutines are substantially more lightweight than threads. In particular, you can easily afford a huge number of concurrently running coroutines since each of them usually only has to keep a relatively compact state and do not need a full-fledged context switching when being suspended or resumed.

The `launch()` builder is suited for cases when the concurrent task is not supposed to compute some result. That's why it takes a `Unit`-typed lambda. If we do need a result, however, there is another builder function which is called `async()`. This function returns an instance of Deferred, a special Job subtype which provides access to the computation result through the `await()` method. When invoked, `await()` suspends until the computation is either completed (thus producing a result) or cancelled. In the latter case, `await()` fails with an exception. You can consider them a non-blocking counterpart of Java futures. For example:

```
import kotlinx.coroutines.*

suspend fun main() {
  val message = GlobalScope.async {
```

```
  delay(100)
   "abc"
  }
  val count = GlobalScope.async {
   delay(100)
   1 + 2
  }
  delay(200)
  val result = message.await().repeat(count.await())
  println(result)
}
```

In this case, we've also marked the `main()` function with suspend to directly call `await()` methods of both the deferred tasks. The output unsurprisingly looks like the following:

abcabcabc

By default, both `launch()` and `async()` builders run coroutines in a shared pool of background threads while the calling thread itself is left unblocked. That's why we had to insert `sleep()` into our `launch()` example as the main thread had little work to do besides waiting for completion of our tasks. The `runBlocking()` builder, on the other hand, creates a coroutine which by default executes in the current thread and blocks until it completes. When the coroutine returns successfully, the return value of the suspend lambda becomes the value of the entire `runBlocking()` call. When a coroutine is cancelled, `runBlocking()` throws an exception. Conversely, when a blocked thread is interrupted, the coroutine started by `runBlocking()` gets cancelled as well. For example:

```
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch {
   delay(100)
   println("Background task: ${Thread.currentThread().name}")
  }
  runBlocking {
   println("Primary task: ${Thread.currentThread().name}")
   delay(200)
  }
}
```

Running this program will produce something like this:

```
Primary task: main
```

```
Background task: DefaultDispatcher-worker-2
```

You can see that the coroutine inside `runBlocking()` is executed in the main thread while the one created by `launch()` gets assigned to a background thread from a shared pool.

Due to its blocking nature, the `runBlocking()` shouldn't be used inside other coroutines. It's intended as a kind of bridge between blocking and non-blocking code and can be used, for example, as a top-level builder in the main function or tests.

Coroutine scopes and structured concurrency

So far, our example coroutines were running in the global scope which effectively means that their lifetime is limited only by that of the entire application. In some cases, we may want to ensure the coroutine execution that is restricted to a particular operation. This is possible thanks to parent-child relationship between concurrent tasks. When you start one coroutine in the context of another, the latter becomes a child of the former. The lifecycles of the parent and child are related so that the parent coroutine may complete only after completion of all its children.

This feature is called a structured concurrency. It can be compared to using blocks and subroutines to constrain a scope of local variables. Let's look at some example:

```
import kotlinx.coroutines.*

fun main() {
 runBlocking {
   println("Parent task started")
   launch {
      println("Task A started")
      delay(200)
      println("Task A finished")
   }
   launch {
      println("Task B started")
      delay(200)
      println("Task B finished")
   }
   delay(100)
   println("Parent task finished")
 }
```

```
  println("Shutting down…")
}
```

The preceding code starts a top-level coroutine which then launches a pair of children by calling launch on the current instance of `CoroutineScope` (which is passed as a receiver to the suspending lambda). If you run this program, you'll see the following result:

```
Parent task started
Task A started
Task B started
Parent task finished
Task A finished
Task B finished
Shutting down…
```

You can see that the main body of the parent coroutine represented by the suspend lambda of the `runBlocking()` call finishes before its children due to having smaller delay of 100 ms. The coroutine itself is not completed at this point and just waits in a suspended state until the completion of both the children. After that parent coroutine completes as well and since we're using the `runBlocking()` builder, here it also unblocks the main thread letting it to print the final message.

You can also introduce a custom scope by wrapping a code block inside the `coroutineScope()` call. Similar to `runBlocking()`, this function returns the value of its lambda and doesn't complete until its children reach completion. The main difference between `coroutineScope()` and `runBlocking()` is that the former is a suspending function which doesn't block the current thread:

```
import kotlinx.coroutines.*

fun main() {
  runBlocking {
    println("Custom scope start")
    coroutineScope {
      launch {
        delay(100)
        println("Task 1 finished")
      }
      launch {
        delay(100)
        println("Task 2 finished")
      }
    }
```

```
    println("Custom scope end")
  }
}
```

Note that the `Custom scope` end message is printed last because the preceding `coroutineScope()` call suspends until both the children finish their execution.

In general, parent-child relationships can form complex coroutine hierarchies which define a shared scope for processing of exceptions and cancellation requests. We'll revisit this topic in the following sections when talking about coroutine jobs and cancelling.

Coroutine context

Each coroutine has an associated context which is represented by the `CoroutineContext` interface and can be accessed by the `coroutineContext` property of the enclosing scope. The context is an immutable collection of key-value pairs which contains various data available to coroutine. Some of them have a special meaning for coroutine machinery and affect how a coroutine gets executed at runtime. The following elements are of particular interest:

- Job which represents the cancellable task performed by the coroutine
- Dispatcher which controls how coroutines are associated with threads

In general, context can store any data implementing `CoroutineContext.Element`. To access a particular element, you can use the `get()` method or the indexing operator supplying a corresponding key:

```
GlobalScope.launch {
  // obtains current job and prints “Task is active: true”
  println("Task is active:
  ${coroutineContext[Job.Key]!!.isActive}")
}
```

By default, coroutines created by standard builders like `launch()` or `async()` inherit their context from the current scope. When necessary, you can supply a different one using the context parameter of the corresponding builder function. To create a new context, you may use the `plus()` function/+ operator which merges data from two contexts together and the `minusKey()` function which removes the element with a given key:

```
import kotlinx.coroutines.*

private fun CoroutineScope.showName() {
 println("Current coroutine:
 ${coroutineContext[CoroutineName]?.name}")
}

fun main() {
 runBlocking {
  showName() // Current coroutine: null
  launch(coroutineContext + CoroutineName("Worker")) {
   showName() // Current coroutine: Worker
  }
 }
}
```

You can also switch context during coroutine execution using the `withContext()` function which takes a new context and a suspending lambda. This can be useful, for example, if you want to run some block of code inside a different thread. We'll see an example of such thread-jumping in the section about coroutine dispatchers.

Coroutine control flow

Job lifecycle

A job is an object which represents the lifecycle of a concurrent task. Using jobs, you can track task states and cancel them when necessary. Possible states of a job are shown in Figure 13.3. Let's take a closer look at what these states mean and how job transition moves from one state into another.

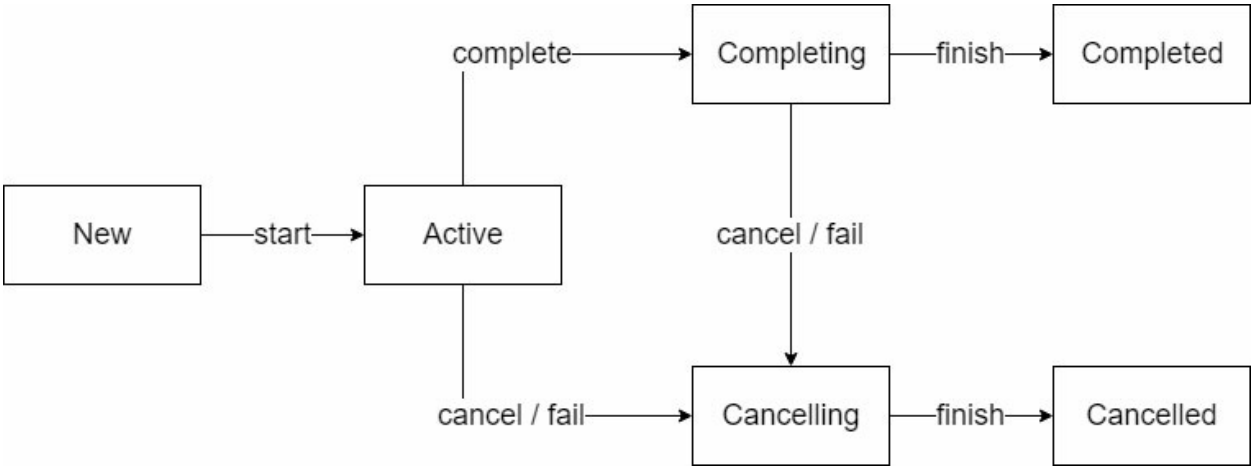

Figure 13.3: *Job states*

An active state means that a job has been started but hasn't yet come to completion. This state is usually used by default; in other words, a job is implicitly started after it's created. Some coroutine builders like `launch()` and `async()` allow you to choose an initial state by specifying an argument of the `CoroutineStart` type:

- `CoroutineStart.DEFAULT` is the default behavior where a job is started immediately.
- `CoroutineStart.LAZY` means that the job is not started automatically; in this case, it's placed into a new state and awaits starting.

A job in the new state can be started by calling its `start()` or `join()` method after which it transitions to the active state. For example:

```
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  val job = launch(start = CoroutineStart.LAZY) {
   println("Job started")
  }
  delay(100)
  println("Preparing to start…")
  job.start()
 }
}
```

The preceding program defers the child coroutine start until the root one prints its message. The output looks like:

```
Preparing to start…
Job started
```

While in the active state, a job can be repeatedly be suspended and resumed by the coroutines machinery. It can also start new jobs which become its children, thus forming a tree-like dependency structure between concurrent computations. You can determine a list of non-completed children jobs using the children property. For example:

```
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  val job = coroutineContext[Job.Key]!!
  launch { println("This is task A") }
```

```
   launch { println("This is task B") }
   // 2 children running
   println("${job.children.count()} children running")
  }
}
```

When a coroutine finishes execution of a suspending lambda block, its job changes its state to completing which basically means waiting for children completion. The job retains this state until all its children complete after which it transitions to the completed state.

You can use Job's `join()` method to suspend the current coroutine until the job in question is complete. The following program ensures that the root coroutine message is printed after its both children finish their execution:

```
import kotlinx.coroutines.*
fun main() {
 runBlocking {
   val job = coroutineContext[Job.Key]!!
   val jobA = launch { println("This is task A") }
   val jobB = launch { println("This is task B") }
   jobA.join()
   jobB.join()
   println("${job.children.count()} children running")
 }
}
```

The resulting output is:

```
This is task A
This is task B
0 children running
```

As expected, there are no active children at the point `job.children.count()` is evaluated.

Cancelling and cancelled states reflect the status of a job whose execution is being/was cancelled either due to unhandled exception, or an explicit call to the `cancel()` method.

The current state of a job can be tracked by its properties: `isActive`, `isCancelled` and `isComplete`. Their meaning can be summarized in the following table. You can also find it in the documentation of the Job interface:

Job state	isActive	isCompleted	isCancelled

New	False	False	False
Active	True	False	False
Completing	True	False	False
Cancelling	False	False	True
Cancelled	False	True	True
Completed	False	True	False

Table 13.1: *Determining the current state by Job properties*

Note that `isCompleted` returns true for both completed and cancelled jobs. You can distinguish between the two by checking the `isCancelled` property. Completed and completing states, on the other hand, are indistinguishable from outside the job itself.

Cancellation

Jobs can be cancelled by calling their `cancel()` method. This provides the standard mechanism for terminating computations which are no longer necessary. The cancellation is cooperative; in other words, a cancellable coroutine itself must check whether its cancellation is requested and respond appropriately. Consider the following program:

```
import kotlinx.coroutines.*

suspend fun main() {
 val squarePrinter = GlobalScope.launch(Dispatchers.Default) {
  var i = 1
   while (true) {
   println(i++)
  }
 }
 delay(100) // let child job run for some time
 squarePrinter.cancel()
}
```

The code starts a coroutine which constantly prints integer numbers. Let it run for about 100 milliseconds and then it tries to cancel. However, if you run the program, you'll find that `squarePrinter` continues to execute. The reason is that it doesn't cooperate in cancellation. One way to fix this is to repeatedly check whether the coroutine was cancelled before doing the next piece of work:

```
import kotlinx.coroutines.*

suspend fun main() {
 val squarePrinter = GlobalScope.launch(Dispatchers.Default) {
 var i = 1
  while (isActive) {
   println(i++)
  }
 }
 delay(100) // let child job run for some time
 squarePrinter.cancel()
}
```

The `isActive` extension property checks whether the current job is in the active state. When applied to `CoroutineScope` (which is passed as a receiver to the coroutine's suspend lambda), it simply delegates to the `isActive` property of the current job. Now, when the parent coroutine calls the `cancel()` method, the state of squarePrinter is changed to cancelling and the next check of the `isActive` condition forces loop termination. When the coroutine finishes its execution, the state is changed to cancelled. If you run the preceding code, you'll see that it terminates after running for approximately 100 milliseconds.

Another solution is to replace the state check with a call with some suspending function which can respond to cancellation by throwing `CancellationException`. This exception is used internally by the coroutines library as a control-flow token signaling that job cancelling is in progress. This is true for all suspending functions such as `delay()` or `join()` defined in the coroutines library. One more example is `yield()` which suspends a given job freeing its thread for other coroutines (similarly to how `Thread.yield()` may suspend the current thread by giving other threads an extra chance to run):

```
import kotlinx.coroutines.*

suspend fun main() {
 val squarePrinter = GlobalScope.launch(Dispatchers.Default) {
  var i = 1
  while (true) {
   yield()
   println(i++)
  }
 }
 delay(100) // let child job run for some time
```

```
  squarePrinter.cancel()
}
```

When a parent coroutine is cancelled, it automatically cancels the execution of all its children and the process continues until the hierarchy is cancelled. Consider the following example:

```
import kotlinx.coroutines.*

fun main() {
  runBlocking {
    val parentJob = launch {
      println("Parent started")
      launch {
        println("Child 1 started")
        delay(500)
        println("Child 1 completed")
      }
      launch {
        println("Child 2 started")
        delay(500)
        println("Child 2 completed")
      }
      delay(500)
      println("Parent completed")
    }
    delay(100)
    parentJob.cancel()
  }
}
```

The program launches a coroutine which then starts a pair of children. All the three tasks are supposed to be delayed for 500 ms before printing the completion message. The parent job, however, is cancelled after 100 ms. As a result, neither it nor its children reach completion and the program output looks like the following:

```
Parent started
Child 1 started
Child 2 started
```

Timeouts

In some cases, we can't wait for completion of a task indefinitely and need to set up some timeout. The coroutines library has a special `withTimeout()`

function exactly for this purpose. For example, the following code starts a coroutine which suspends while the file is being read:

```
import kotlinx.coroutines.*
import java.io.File

fun main() {
 runBlocking {
  val asyncData = async { File("data.txt").readText() }
  try {
   val text = withTimeout(50) { asyncData.await() }
   println("Data loaded: $text")
  } catch (e: Exception) {
   println("Timeout exceeded")
  }
 }
}
```

If the file gets read within 50 milliseconds, `withTimeout()` just returns the result of its block. Otherwise, it throws `TimeoutCancellationException` (which is a subclass of `CancellationException`) and the reading coroutine is cancelled.

There is also a similar function `withTimeoutOrNull()` which doesn't throw an exception when timeout is exceeded and simply returns null instead.

Coroutine dispatching

While coroutines give you a thread-independent way to implement suspendable computations, they still need to be associated with some thread(s) when run. The coroutines library includes a special component whose task is to control the threads that are used to execute a particular coroutine. This component is called a coroutine dispatcher.

A dispatcher is a part of the coroutine context so you can specify it in the coroutine builder functions like `launch()` and `runBlocking()`. Since the dispatcher is also a single-element context by itself, you can simply pass it to a coroutine builder:

```
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  // running coroutine using global thread pool dispatcher
  launch(Dispatchers.Default) {
```

```
    println(Thread.currentThread().name) // DefaultDispatcher-
    worker-1
   }
  }
}
```

Coroutine dispatchers are somewhat similar to Java executors which distribute threads between a set of parallel tasks. In fact, you can easily convert the existing implementation of Executor into a respective coroutine dispatcher using the `asCoroutineDispatcher()` extension function. In the following example, we create a pool-based executor service with the custom thread factory which assigns names like WorkerThread1, `WorkerThread2`, … to executor threads. Then, we convert it into a dispatcher and use it to start several coroutines in parallel. Note that we explicitly set up worker threads as daemons so that they don't prevent the program termination after all the coroutines complete:

```
import kotlinx.coroutines.*
import java.util.concurrent.ScheduledThreadPoolExecutor
import java.util.concurrent.atomic.AtomicInteger

fun main() {
 val id = AtomicInteger(0)
 val executor = ScheduledThreadPoolExecutor(5) { runnable ->
  Thread(
   runnable,
   "WorkerThread-${id.incrementAndGet()}"
  ).also { it.isDaemon = true }
 }
 executor.asCoroutineDispatcher().use { dispatcher ->
  runBlocking {
   for (i in 1..3) {
    launch(dispatcher) {
     println(Thread.currentThread().name)
     delay(1000)
    }
   }
  }
 }
}
```

The delay forces the executor to create separate threads so the preceding code would print the following:

```
WorkerThread-1
WorkerThread-2
```

```
WorkerThread-3
```

The specific thread order may vary, though.

Note that when invoked on an instance of `ExecutorService`, `asCoroutineDispatcher()` returns `ExecutorCoroutineDispatcher` which also implements the `Closeable` interface. You need to use the `close()` method to shut down the underlying executor service and free system resources allocated to maintain its threads or wrap dispatcher usages inside the `use()` function block like we did in the preceding example.

The coroutines library also comes with a set of out-of-the box dispatcher implementations. Some of them can be accessed via the `Dispatchers` object:

- `Dispatchers.Default`: A shared thread pool whose size is by default equal to the number of available CPU cores or 2 (whatever is greater). This implementation is generally suited for CPU-bound computations where the task performance is limited primarily by the CPU speed.
- `Dispatchers.IO`: A similar implementation based on a thread pool which is optimized for running potentially blocking I/O-intensive tasks such as reading/writing files. This dispatcher shares the thread pool with default implementation by adding or terminating extra threads when necessary.
- `Dispatchers.Main`: A dispatcher which operates exclusively in the UI event thread where the user input is processed.

It’s also possible to create a dispatcher based on a private thread pool or even a single thread using either the `newFixedThreadPoolContext()` or `newSingleThreadPoolContext()` function. For example, we can rewrite our sample based on `Executor` as:

```
import kotlinx.coroutines.*

@Suppress("EXPERIMENTAL_API_USAGE")
fun main() {
 newFixedThreadPoolContext(5, "WorkerThread").use { dispatcher -
 >
  runBlocking {
   for (i in 1..3) {
    launch(dispatcher) {
     println(Thread.currentThread().name)
     delay(1000)
    }
```

```
    }
   }
  }
}
```

Note that we've used the `@Suppress` annotation because `newFixedThreadPoolContext()` and `newSingleThreadPoolContext()` are currently marked as obsolete API and are expected to be replaced by newer functions based on a shared thread pool.

When the dispatcher is not specified explicitly (like we did in earlier examples), it's automatically inherited from the scope you use to start a coroutine. Consider the following example:

```
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  println("Root: ${Thread.currentThread().name}")
  launch {
   println("Nested, inherited: ${Thread.currentThread().name}")
  }
  launch(Dispatchers.Default) {
   println("Nested, explicit: ${Thread.currentThread().name}")
  }
 }
}
```

We start a top-level coroutine which runs in the main thread and launches two nested coroutines: one whose context (and coroutine dispatcher as a result) is inherited from the parent coroutine and another where the dispatcher is specified explicitly. Thus, the preceding code would print the following:

```
Root: main
Nested, explicit: DefaultDispatcher-worker-1
Nested, inherited: main
```

In the absence of a parent coroutine, the dispatcher is implicitly assumed to be `Dispatchers.Default` except for the `runBlocking()` builder which is confined to the current thread.

The coroutine need not have the same dispatcher throughout its entire lifetime. Since the dispatcher is a part of coroutine context, it can be overridden using the `withContext()` function:

```
import kotlinx.coroutines.*
```

```
@Suppress("EXPERIMENTAL_API_USAGE")
fun main() {
 newSingleThreadContext("Worker").use { worker ->
 runBlocking {
  println(Thread.currentThread().name) // main
  withContext(worker) {
   println(Thread.currentThread().name) // Worker
   }
   println(Thread.currentThread().name) // main
  }
 }
}
```

This technique comes in handy when we want to confine an execution of a particular routine fragment to a single thread.

Exception handling

When it comes to exception handling, various coroutine builders follow one of the two basic strategies. The first one implemented by builders like `launch()` is to propagate an exception to the parent coroutine. In this case, the execution proceeds as follows:

- The parent coroutine is cancelled with the same exception as a cause. This causes it to cancel all the remaining children.
- When children are cancelled, the parent passes an exception to further up the coroutine tree.

The process continues until it reaches a coroutine with a global scope. After that, it's handled by `CoroutineExceptionHandler`. Consider, for example, the following program:

```
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  launch {
   throw Exception("Error in task A")
   println("Task A completed")
  }
  launch {
   delay(1000)
   println("Task B completed")
  }
  println("Root")
```

```
  }
}
```

The top-level coroutine starts a pair of nested tasks with the first one throwing an exception. This causes cancellation of the root task and both of its children and since no custom handler is provided, the program falls back to the default behavior represented by `Thread.uncaughtExceptionHandler`. As a result, it would print followed by the exception stack trace:

```
Root
Exception in thread "main" java.lang.Exception: Error in task A
```

`CoroutineExceptionHandler` defines a single method which takes the current coroutine context and a thrown exception:

```
fun handleException(context: CoroutineContext, exception:
Throwable)
```

The simplest way to construct a handler is to use the `CoroutineExceptionHandler()` function which takes a two-argument lambda:

```
val handler = CoroutineExceptionHandler{ _, exception ->
 println("Caught $exception")
}
```

To configure its instance for processing exceptions, you can put it into the coroutine context. Since the handler is a trivial context by itself, you can just pass it as the context argument into the coroutine builder:

```
import kotlinx.coroutines.*

suspend fun main() {
 val handler = CoroutineExceptionHandler{ _, exception ->
  println("Caught $exception")
 }

 GlobalScope.launch(handler) {
  launch {
   throw Exception("Error in task A")
   println("Task A completed")
  }
  launch {
   delay(1000)
   println("Task B completed")
  }
  println("Root")
```

```
  }.join()
}
```

Now the program prints the following, thus, overriding the default behavior:

```
Root
Caught java.lang.Exception: Error in task A
```

When no handler instance is defined in the context, the coroutines library will invoke all global handlers configured via the JVM `ServiceLoader` mechanism as well as `uncaughtExceptionHandler` for the current thread.

Note that `CoroutineExceptionHandler` can only be specified for a coroutine launched in the global scope and is used only for its children. That's why we had to replace `runBlocking()` with `GlobalScope.launch()` and mark the `main()` function with suspend to make use of the suspending `join()` call. If we retain `runBlocking()` from our original example, but supply it with a handler. The program would still use the default exception handler since our coroutines wouldn't be run in the global scope:

```
import kotlinx.coroutines.*

fun main() {
  val handler = …
  runBlocking(handler) {
    …
  }
}
```

Another way to handle exception used by `async()` builder is to preserve the thrown exception and rethrow it later when the corresponding `await()` is called. Let's modify our example slightly:

```
import kotlinx.coroutines.*

fun main() {
  runBlocking {
    val deferredA = async {
      throw Exception("Error in task A")
      println("Task A completed")
    }
    val deferredB = async {
      println("Task B completed")
    }
    deferredA.await()
    deferredB.await()
    println("Root")
```

```
   }
}
```

Now, the output looks like the following:

```
Exception in thread "main" java.lang.Exception: Error in task A
```

The reason is that exception is rethrown by `deferredA.await()`, so the program fails to reach the `println("Root")` statement.

Note that async-like builders, which rethrow an exception when you access coroutine data, do not rely on `CoroutineExceptionHandler`. So even if you have its instance preconfigured in the coroutine context, it has no effect (just as we've seen in the `runBlocking()` example). The program will still fall back to the default handler.

What if we want to process exceptions thrown by the nested coroutines at the level of their parent without relying on global handlers? Let's see what happens if we attempt to process rethrown exceptions using the `try-catch` block:

```
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  val deferredA = async {
     throw Exception("Error in task A")
   println("Task A completed")
  }
  val deferredB = async {
   println("Task B completed")
  }
  try {
   deferredA.await()
   deferredB.await()
  } catch (e: Exception) {
   println("Caught $e")
  }
  println("Root")
 }
}
```

If you run this code, you'll see that the handle is indeed activated, but the program still fails with an exception:

```
Caught java.lang.Exception: Error in task A
Root
```

```
Exception in thread "main" java.lang.Exception: Error in task A
```

The reason is that the exception is rethrown automatically to cancel the parent coroutine when its child (task A in this case) fails. To override this behavior, we can use a so called supervisor job.

With supervisor jobs, cancellation propagates only in the downward direction. If you cancel a supervisor, it automatically cancels all its children, but if a child is cancelled instead, the supervisor and its remaining children remain active.

To convert the parent coroutine into a supervisor, we define a new scope using the `supervisorScope()` function instead of `coroutineScope()`. Let's modify our previous example:

```
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  supervisorScope {
   val deferredA = async {
    throw Exception("Error in task A")
    println("Task A completed")
   }
   val deferredB = async {
    println("Task B completed")
   }
   try {
    deferredA.await()
   } catch (e: Exception) {
    println("Caught $e")
   }
   deferredB.await()
   println("Root")
  }
 }
}
```

Now, the exception is not rethrown after processing and both task B and root coroutine reach completion:

```
Task B completed
Caught java.lang.Exception: Error in task A
Root
```

Note that the supervisor behavior extends to normal cancellations as well: calling `cancel()` on one of its children jobs doesn't cause cancellation of its

siblings or supervisor itself.

Concurrent communication

In this section, we'll talk about more advanced features of the coroutines library which allows you to efficiently share data between multiple concurrent tasks while retaining thread safety. To be exact, we'll focus on channels which provide a mechanism for passing data streams between coroutines and actors which allow you to safely share a mutable state without any synchronizations and locks.

Channels

Channels offer you a convenient way to share an arbitrary data stream between coroutines. The basic operations on any `channel` represented by the `Channel` interface is sending data elements by the `send()` method and receiving them by the `receive()` method, respectively. When these methods can't complete their work; for example, when the channel's internal buffer is full and you try to send data to it, they suspend the current coroutine and resume them later when it's possible. That's the major difference between channels and blocking queues which play a similar role in Java's concurrency API but work by blocking the calling thread.

Channels can be constructed by the generic `Channel()` function which takes an integer value describing the channel capacity. One of the basic implementations is a channel with internal buffer of a limited size. When the buffer is full, a call to `send()` is suspended until at least one element is received. Similarly, a call to `receiver()` is suspended when the buffer is empty until at least one element gets sent. Let's consider an example:

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.*
import kotlin.random.Random

fun main() {
 runBlocking {
  val streamSize = 5
  val channel = Channel<Int>(3)
  launch {
   for (n in 1..streamSize) {
    delay(Random.nextLong(100))
    val square = n*n
```

```
      println("Sending: $square")
      channel.send(square)
     }
    }
    launch {
     for (i in 1..streamSize) {
      delay(Random.nextLong(100))
      val n = channel.receive()
      println("Receiving: $n")
     }
    }
  }
}
```

The first coroutine produces a stream of integer squares and sends them to the channel which can hold up to three elements while the second one concurrently receives the generated numbers. We've inserted random delays to provoke occasional suspension when either of the coroutines don't catch up with its counterpart leading to empty/full channel buffer. A possible result may look like this:

```
Sending: 1
Receiving: 1
Sending: 4
Receiving: 4
Sending: 9
Sending: 16
Receiving: 9
Sending: 25
Receiving: 16
Receiving: 25
```

Although, the output may vary depending on actual delays and other circumstances, channels ensure that all values are received in the same order as they are being sent.

The `Channel()` function can also take some special values which produce channels with different behavior. These values are represented by constants in the companion object of the `Channel` interface:

- `Channel.UNLIMITED (= Int.MAX_VALUE)`: This is a channel with unlimited capacity whose internal buffer grows on demand. Such channels never suspend on `send()`, but can suspend on `receiver()` when the buffer is empty.

- `Channel.RENDEZVOUS (= 0)`: This is arendezvous channel which has no internal buffer. Any call to `send()` suspends until some other coroutine invokes `receive()`. Similarly, the `receive()` call is suspended until someone invokes `send()`. This channel is created by default when you omit the capacity argument.
- `Channel.CONFLATED (= -1)`: This is a conflated channel which stores at most one element which is overwritten by `send()` so that any unread sent values are lost. In this case, the `send()` method never suspends.
- Any positive value less than `UNLIMITED` produces a channel with limited-size buffer.

The rendezvous channel ensures that producer and consumer coroutines are activated in turns. For example, if we change our earlier example by setting the channel capacity to zero, we'll always get a stable operation order regardless of delays:

```
Sending: 1
Receiving: 1
Sending: 4
Receiving: 4
Sending: 9
Receiving: 9
Sending: 16
Receiving: 16
Sending: 25
Receiving: 25
```

Conflated channels can be used if you don't need every element in a stream and can afford discarding some of them if the consume routine doesn't catch up with the producer. Let's modify our first example by setting the consumer delay to be twice as much as the producer's:

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  val streamSize = 5
  val channel = Channel<Int>(Channel.CONFLATED)
  launch {
   for (n in 1..streamSize) {
    delay(100)
    val square = n*n
```

```
      println("Sending: $square")
      channel.send(square)
     }
    }
    launch {
     for (i in 1..streamSize) {
      delay(200)
      val n = channel.receive()7
      println("Receiving: $n")
     }
    }
  }
}
```

As a result, only about half of produced values are received and processed. A possible output may look like the following:

```
Sending: 1
Receiving: 1
Sending: 4
Sending: 9
Receiving: 9
Sending: 16
Sending: 25
Receiving: 25
```

If you run the preceding program, you'll also see that it doesn't terminate after printing the last line. The reason is that our receiver expects to get at least five values since we're iterating from `1` to `streamSize`. But since only about `streamSize/2` values are actually received, this condition can never be satisfied. What we need in this situation is some kind of a signal which would mean that the channel is closed and won't send any further data. The `Channel` API allows you to do that by calling the `close()` method on the producer side. On the consumer side, we can replace the fixed-number loop with iteration over channel data:

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.*

fun main() {
 runBlocking {
  val streamSize = 5
  val channel = Channel<Int>(Channel.CONFLATED)
  launch {
   for (n in 1..streamSize) {
    delay(100)
```

```
      val square = n*n
      println("Sending: $square")
      channel.send(square)
     }
     channel.close()
    }
    launch {
     for (n in channel) {
      println("Receiving: $n")
      delay(200)
     }
    }
  }
}
```

Now, the program terminates after the data exchange is complete.

On the consumer side, you can also use the `consumeEach()`function to read all channel content instead of explicit iteration:

```
channel.consumeEach {
 println("Receiving: $n")
 delay(200)
}
```

After the channel is closed, any attempt to call `send()` will fail with `ClosedSendChannelException`. Calls to `receive()` will return unread elements until the channel is exhausted after which they will throw `ClosedSendChannelException` as well.

Channel communication does not necessarily involve just a single producer and a single consumer. For example, the same channel can be concurrently read by multiple coroutines. This is called fanning out:

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.*
import kotlin.random.Random

fun main() {
 runBlocking {
  val streamSize = 5
  val channel = Channel<Int>(2)
  launch {
   for (n in 1..streamSize) {
    val square = n*n
    println("Sending: $square")
    channel.send(square)
   }
```

```
    channel.close()
   }
   for (i in 1..3) {
    launch {
     for (n in channel) {
      println("Receiving by consumer #$i: $n")
      delay(Random.nextLong(100))
     }
    }
   }
  }
}
```

Data stream generated by a producer coroutine is split between three consumers. A possible output can look like this:

```
Sending: 1
Sending: 4
Sending: 9
Receiving by consumer #1: 1
Receiving by consumer #2: 4
Receiving by consumer #3: 9
Sending: 16
Sending: 25
Receiving by consumer #3: 16
Receiving by consumer #1: 25
```

Similarly, we can fan in by collecting output of multiple producers in the same channel and feeding it into a single consumer coroutine. In a more general case, any number of producers and consumers can communicate via multiple channels. In general, the channel behavior is fair with respect to multiple coroutines in a sense that a coroutine, which gets to invoke `receive()` first, gets the next element.

Producers

There is a special `producer()` coroutine builder which allows you to construct concurrent data stream similar to the `sequence()` function we've discussed earlier when talking about the collection API. This builder introduces `ProducerScope` which provides the `send()` method similar to a channel:

```
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.*
```

```
fun main() {
 runBlocking {
  val channel = produce {
   for (n in 1..5) {
    val square = n*n
    println("Sending: $square")
    send(square)
   }
  }
  launch {
   channel.consumeEach { println("Receiving: $it") }
  }
 }
}
```

Note that you do not need to explicitly close a channel in this case. The `producer()` builder will do it automatically on coroutine termination.

In terms of exception handling, `produce()` follows the policy of `async()/await()`; an exception thrown inside `produce()` is preserved and rethrown in the first coroutine which invokes the channel's `receive()`.

Tickers

The `coroutines` library has a special variety of a rendezvous channel which is called a ticker. This channel produces a stream of `Unit` values with a given delay between subsequent elements. To construct it, you can use the `ticker()` function which allows you to specify the following:

- `delayMillis`: The delay in milliseconds between ticker elements.
- `initialDelayMillis`: The delay before producing the first element; by default, it's the same as `delayMillis`.
- `context`: Coroutine context in which the ticker is supposed to run (empty by default).
- `mode`: A value of `TickerMode` enum which determines a mode of ticker behavior:
    - `TickerMode.FIXED_PERIOD`: A ticker will choose the delay to maintain a constant period between element generations as much as possible.
    - `TickerMode.FIXED_RATE`: A ticker will simply make a specified

delay before sending each element regardless of how much time has passed since the last receive.

To see the difference between ticker modes, let's consider the following code:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
 val ticker = ticker(100)
 println(withTimeoutOrNull(50) { ticker.receive() })
 println(withTimeoutOrNull(60) { ticker.receive() })
 delay(250)
 println(withTimeoutOrNull(1) { ticker.receive() })
 println(withTimeoutOrNull(60) { ticker.receive() })
 println(withTimeoutOrNull(60) { ticker.receive() })
}
```

When run, it produces the following output:

```
null
kotlin.Unit
kotlin.Unit
kotlin.Unit
null
```

Let's see how its execution proceeds step-by-step:

1. We try to receive the ticker signal within 50 ms timeout. Since the ticker delay is 100 ms, `withTimeOutOrNull()` returns null as no signal was sent yet.
2. Then, we try to receive the signal within the next 60 ms. This time we'll certainly get a non-null result because at least 100 ms will pass since the ticker has started. Once the `receive()` is called the ticker will resume.
3. Then, the consumer coroutine is suspended for about 250 ms. 100 ms later, the ticker sends another signal and suspends waiting for it to be received. After that, both coroutines remain in a suspended state for 150 ms.
4. The consumer coroutine resumes and tries to request the signal. Since the signal was already sent, `receive()` returns immediately (thus we can set a small timeout of 1 ms) letting the ticker coroutine to resume. Now, the ticker will measure the time elapsed since the last signal was

sent and find that it’s about 250 ms. This interval contains two whole delays (200 ms) and a remainder of about 50 ms. The ticker then adjust its own waiting time before the next signal to be 100 – 50 = 50 ms so that the signal is sent when the whole delay (100 ms) is passed.

5. The consumer tries to receive the single within 60 ms timeout and may certainly succeed since the next signal should be sent in less than 50 ms.
6. The last attempt to receive the signal happens almost immediately, so the ticker will wait for the whole delay (100 ms) again. As a result, the last call to `receive()` returns null because the signal won’t be received within 60 ms timeout.

If we set the ticker mode to `FIXED_RATE`, the result will change:

```
null
kotlin.Unit
kotlin.Unit
null
kotlin.Unit
```

The first execution proceeds in almost the same way. The difference comes after the consumer coroutine resumes after making a long delay of 250 ms. The third `receive()` also returns immediately since the ticker has already sent its signal within that 250 ms period, but now, it won’t take the elapsed time into account and simply wait for another 100 ms. As a result, the fourth call to `receive()` returns null because the signal is not sent yet after 60 ms. At the moment of the fifth call, however, this interval rises above 100 ms and the signal is received.

Note that the ticker-related API is currently considered experimental and may be replaced in future versions of the coroutines library.

Actors

A common way to implement a thread-safe access to a shared mutable state is given by the actor model. An actor is an object which comprises some internal state and means to concurrently communicate with the other actor by sending messages. Actors listen for incoming messages and can respond to them by modifying their own state, sending more messages, and starting new actors. An actor’s state is private so other actors can’t use it directly; it can only be accessed by sending a message, thus relieving you from the need to

use lock-based synchronization.

In Kotlin coroutines library, actors can be created using the `actor()` coroutine builder. It introduces a special scope (`ActorScope`) which combines basic coroutine scope with a receiver channel you can to access incoming messages. This builder is somewhat similar to `launch()` since it also starts a job which isn't meant to produce the result by itself and follows the same exception handling policy as `launch()` coroutine builders relying on `CoroutineExceptionHandler`.

To demonstrate the basic usages of actors API, let's consider a simple example of an actor which maintains a bank account and can withdraw/deposit a given amount of money. First, we need to define a set of classes which represent incoming messages:

```
sealed class AccountMessage
class GetBalance(
 val amount: CompletableDeferred<Long>
) :AccountMessage()
class Deposit(val amount: Long) : AccountMessage()
class Withdraw(
 val amount: Long,
 val isPermitted: CompletableDeferred<Boolean>
) :AccountMessage()
```

Using the sealed class hierarchy will allow us to employ exhaustive `when` expressions for instances of `AccountMessage` class.

Note that the `GetBalance` instance has a property of the `CompletableDeferred` type. Our actor will use this property to send the current account balance back to the coroutine which requests it using the `GetBalance` message. Similarly, the Withdraw class has the `isPermitted` property which will receive true if the withdraw is successful and false otherwise.

Now, we can get to implement the actor responsible for maintaining the account balance. The basic logic is simple; we continuously poll the incoming channel and perform one of the possible actions depending on the received message:

```
fun CoroutineScope.accountManager(
 initialBalance: Long
) = actor<AccountMessage> {
 var balance = initialBalance
```

```
  for (message in channel) {
    when (message) {
      is GetBalance ->message.amount.complete(balance)
      is Deposit -> {
        balance += message.amount
        println("Deposited ${message.amount}")
      }
      is Withdraw -> {
      val canWithdraw = balance >= message.amount
        if (canWithdraw) {
          balance -= message.amount
          println("Withdrawn ${message.amount}")
        }
        message.isPermitted.complete(canWithdraw)
      }
    }
  }
}
```

The `actor()` builder can be as a counterpart of `produce()`; both rely on channels for communication but while actors use them for receiving data, producers create channels for sending data to their consumers. By default, actors use rendezvous channels, but you can change it by specifying the capacity argument in the `actor()` function call.

Note the use of `complete()` method on `CompletableDeferred`. That's how we send the request result back to the actor client.

Now, let's add a pair of coroutines which communicate without the actor:

```
private suspend fun SendChannel<AccountMessage>.deposit(
  name: String,
  amount: Long
) {
  send(Deposit(amount))
  println("$name: deposit $amount")
}

private suspend fun SendChannel<AccountMessage>.tryWithdraw(
  name: String,
  amount: Long
) {
  val status = CompletableDeferred<Boolean>().let {
    send(Withdraw(amount, it))
    if (it.await()) "OK" else "DENIED"
  }
  println("$name: withdraw $amount ($status)")
}
```

```
private suspend fun SendChannel<AccountMessage>.printBalance(
 name: String
) {
 val balance = CompletableDeferred<Long>().let {
  send(GetBalance(it))
  it.await()
 }
 println("$name: balance is $balance")
}
fun main() {
 runBlocking {
  val manager = accountManager(100)
  withContext(Dispatchers.Default) {
   launch {
    manager.deposit("Client #1", 50)
    manager.printBalance("Client #1")
   }
   launch {
    manager.tryWithdraw("Client #2", 100)
    manager.printBalance("Client #2")
   }
  }
  manager.tryWithdraw("Client #0", 1000)
  manager.printBalance("Client #0")
  manager.close()
 }
}
```

To send the actor a message, we use the `send()` method provided by the corresponding channel. Here is an example of the possible output:

```
Client #1: deposit 50
Deposited 50
Withdrawn 100
Client #2: withdraw 100 (OK)
Client #2: balance is 50
Client #1: balance is 50
Client #0: withdraw 1000 (DENIED)
Client #0: balance is 50
```

Although the operation order may vary (especially when it comes to parallel execution), the results remain consistent. We don’t need any synchronization primitives like locks or critical sections since there is no publicly accessible mutable state.

One more thing worth noting is that actor `builders()` are currently considered an experimental API which is subject to possible changes in

future.

Using Java concurrency

Apart from the Kotlin-specific coroutines library, you can also make use of the JDK concurrency API when targeting the JVM platform. In this section, we'll discuss some helper functions provided by the Kotlin standard library to simplify common concurrency-related tasks such as creating threads and synchronization.

Starting a thread

To start a general-purpose thread, you can use the thread() function which allows you to specify both a thread runnable in the form of Kotlin lambda as well as a set of basic thread properties:

- `start`: Whether a thread is started once it gets created (true by default)
- `isDaemon`: Whether a thread is started in the daemon mode (false by default). Daemon threads do not prevent JVM termination and thus are shut down automatically on termination of the main thread.
- `contextClassLoader`: The custom class loader which is used by the thread code to load classes and resources (null by default).
- `name`: This is the custom thread name. By default, it's null which means that the name is chosen automatically (in the form of `Thread-1`, `Thread-2`, etc.)
- `priority`: Thread priority which ranges from `Thread.MIN_PRIORITY (= 1)` to `Thread.MAX_PRIORITY (= 10)`and affects how much CPU time the thread will get as compared to others. By default, it's equal to -1 which means that priority is chosen automatically.
- `block`: A function value of type `() -> Unit` which is run in the new thread.

For example, the following program starts a thread which prints a message every 150 milliseconds:

```
import kotlin.concurrent.thread

fun main() {
 println("Starting a thread…")
```

```
  thread(name = "Worker", isDaemon = true) {
    for (i in 1..5) {
      println("${Thread.currentThread().name}: $i")
      Thread.sleep(150)
    }
  }
  Thread.sleep(500)
  println("Shutting down…")
}
```

Since a new thread is started as a daemon, it only manages to print its message four times because JVM terminates once the main thread finishes execution after 500 ms of sleep. As a result, the program output looks as follows:

```
Starting a thread…
Worker: 1
Worker: 2
Worker: 3
Worker: 4
Shutting down…
```

Another group of functions is related to Java timers which allow you to concurrently execute some periodic actions at the specific time. The `timer()` function schedules a timer which runs some tasks with a fixed delay relative to the time of its last execution. As a result, when some execution takes more time, all subsequent runs are postponed. In this sense, it can be compared to a Kotlin ticker working in the `FIXED_RATE` mode. When configuring a timer with a `timer()` call, you can specify the following options:

- `name`: Name of the timer thread (null by default)
- `daemon`: Whether the time thread is run as daemon (false by default)
- `startAt`: The `Date` object describing when the first time event should happen
- `period`: Desired number of milliseconds between successive timer executions
- `action: TimeTask.() -> Unit` lambda which is run on each timer's execution

Alternatively, you can use another `timer()` overload with the `initalDelay` parameter which specifies the moment of the first event as delay from the current time (defaulting to zero).

Let’s rewrite our previous example using timers:

```
import kotlin.concurrent.timer

fun main() {
 println("Starting a thread…")
 var counter = 0
 timer(period = 150, name = "Worker", daemon = true) {
  println("${Thread.currentThread().name}: ${++counter}")
 }
 Thread.sleep(500)
 println("Shutting down…")
}
```

There is also a similar pair of `fixedRateTimer()` functions which sets up a timer with a fixed delay between start of subsequent executions. It can be compared to a ticker in `FIXED_PERIOD` mode which tries to compensate additional delays to ensure a constant period of timer events in the long run.

Synchronization and locks

Synchronization is a common primitive which ensures that the specific code fragment is executed in a single thread. When such a fragment is already being executed in some threads, any other threads trying to enter it are forced to wait. In Java, there are two ways of introducing synchronization into your code. First, you can wrap it inside a special synchronized block specifying some object which acts as a lock. In Kotlin, the syntax is quite similar, although you use a standard library function with a lambda rather than a built-in language structure:

```
import kotlin.concurrent.thread

fun main() {
 var counter = 0
 val lock = Any()
 for (i in 1..5) {
  thread(isDaemon = false) {
   synchronized(lock) {
    counter += i
    println(counter)
   }
  }
 }
}
```

Although the order of individual additions may vary thus giving different intermediate results, synchronization ensures that the total sum is always equal to 15. A possible output will look like the following:

```
1
4
8
13
15
```

In general, the `synchronized()` function returns the value of its lambda. For example, we can use it to retrieve one of the intermediate counter values at the moment of the call:

```
import kotlin.concurrent.thread

fun main() {
 var counter = 0
 val lock = Any()
 for (i in 1..5) {…}
 val currentCounter = synchronized(lock) { counter }
 println("Current counter: $currentCounter")
}
```

While the result may vary, it is always equal to some intermediate values produced by one of the five adder threads.

Another way you can use in Java is to mark a method with a synchronized modifier; in this case, the entire method body is considered synchronized with respect to the current instance of the containing class or `Class` instance itself (if the method in question is static). In Kotlin, you use the `@Synchronized` annotation for the same purpose:

```
import kotlin.concurrent.thread

class Counter {
 private var value = 0

 @Synchronized fun addAndPrint(value: Int) {
  value += value
  println(value)
 }
}

fun main() {
 val counter = Counter()
 for (i in 1..5) {
```

```
    thread(isDaemon = false) { counter.addAndPrint(i) }
  }
}
```

The standard library also includes the `withLock()` function which allows you to execute some lambda under a given `Lock` object (from the `java.util.concurrent.locks` package), which is similar to the synchronized block. In this case, you don't need to worry about releasing your lock on exception because this is handled by `withLock()` itself. As an example, let's apply it to our Counter class:

```
class Counter {
  private var value = 0
  private val lock = ReentrantLock()

  fun addAndPrint(value: Int) {
    lock.withLock {
      value += value
      println(value)
    }
  }
}
```

On top of it, there are `read()` and `write()` functions which execute a given action under the read/write locks of the `ReentrantReadWriteLock` object. The `write()` function also extends on `ReentrantReadWriteLock` semantics by supporting automatic upgrade of the existing read lock to write one.

Java vs. Kotlin: Note that `wait()`, `notify()`, and `notifyAll()` methods defined by Java's Object class are not available for Kotlin's `Any`. If necessary you can, however, use them by explicitly casting a value to `java.lang.Object`:

```
(obj as Object).wait()
```

Bear in mind that `wait()`, like other blocking method shouldn't be used inside suspending functions.

Conclusion

In this chapter, we learned about the fundamentals of coroutine-based concurrency in Kotlin. We looked at how the concurrent code can be made of suspending functions and coroutine builders, how to manage coroutine lifetime using contexts and scopes. We also discussed the cooperative

cancellation and exception handling mechanisms and examined the lifecycle of concurrent tasks. We also learned how to use channel and actor-based communication for efficient sharing of data between multiple concurrent tasks.

As an extra topic, we looked at some helpful functions the Kotlin standard library provides to utilize the concurrency API available on the JVM platform.

In the next chapter, we'll focus on the subject of testing. We'll discuss several Kotlin-aware frameworks and see how Kotlin features and DSLs can help us in writing various kinds of test cases.

Questions

1. What is a suspending function? How does its behavior differ from that of an ordinary function?
2. How do you create coroutines with `launch()` and `async()` builders. What's the difference?
3. Explain the purpose of the `runBlocking()` builder.
4. What is a structured concurrency?
5. Describe the lifecycle of a coroutine job. How job cancellation is propagated in a coroutine tree?
6. What is a coroutine dispatcher? Describe common dispatcher implementations provided by the coroutines library.
7. How can you change a dispatcher from inside the coroutine?
8. Describe exception handling mechanisms used by the coroutines library. What is a purpose of `CoroutineExceptionHandler`?
9. What is a supervisor job? How can you use it to handle exceptions thrown by nested coroutines?
10. What is a channel? What kinds of channels are supported by the `coroutines` library?
11. How can you build a channel with the `produce()` function?
12. Describe the behavior of ticker channels.
13. Describe an idea of the actor model. How can you use actors in the

Kotlin `coroutines` library?

14. Which utilities does the Kotlin standard library provide for creating threads?
15. How can you use thread synchronization and locks in the Kotlin code?

CHAPTER 14

Testing with Kotlin

Testing frameworks constitutes a major part of software development ecosystem. They help creating of reusable test code which helps to maintain software quality throughout the development lifecycle. Thanks to a well-designed Java interoperability, Kotlin developers can benefit from numerous testing tools targeting the JVM platform such as JUnit, TestNG, Mockito, and others.

The Kotlin ecosystem, however, has given rise to some frameworks which specifically targets Kotlin developers by utilizing powerful features of the language to create concise and expressive test code. In this chapter, we'll focus on the KotlinTest, a powerful open-source testing framework developed at

http://github.com/kotlintest/kotlintest. We'll take a look at the following three main topics:

- How to use organize test code using KotlinTest specification styles?
- How to express various test assertions using matchers, inspectors, and autogenerated data sets for property-based testing and so on?
- How to ensure correct initialization and finalization of a test environment as well as provide a test configuration?

We'll start with explaining how to configure KotlinTest for use in IntelliJ IDEA projects.

Structure

- KotlinTest specifications
- Assertions
- Fixtures and configurations

Objective

Learn to write test specifications using features provided by the KotlinTest framework.

KotlinTest specifications

In this section, we'll talk about how to configure KotlinTest for use in IntelliJ IDEA projects and different test layouts provided by this testing framework. All examples presented in the chapter will use KotlinTest 3.3.

Getting started with KotlinTest

In order to use KotlinTest, we need to add the project dependencies. We've already seen how to add an external dependency to the IntelliJ IDEA project in *Chapter 13, Concurrency* when discussing the Kotlin coroutines library. Adding test framework is basically similar. First, we add a library in the Project Structure dialog using its Maven coordinates `io.kotlintest:kotlintest-runner-junit5:3.3.0` (see *Figure 14.1*).

If you're using a build automation system like Maven or Gradle, you can configure KotlinTest by adding its dependency to the corresponding buildfile.

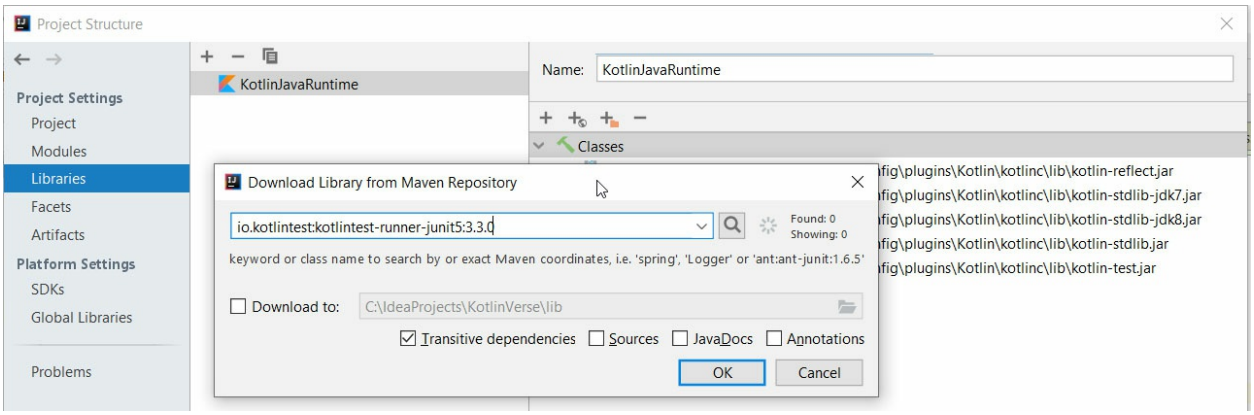

Figure 14.1: *Adding KotlinTest library*

After that IDE will suggest you to add a new library to modules of your project. The next step is to set a dependency scope. Switch to the **`Modules`** view on the left, select a module of interest and open the **`Dependencies`** tab. You'll see the newly added library added to dependencies with its scope set to **`Compile`**. It means that the library will be included to the classpath of both

production and test sources during their compilation and running in the IDE. Since we need KotlinTest only for testing purposes, the scope should be changed to **`Test`** (*Figure 14.2*):

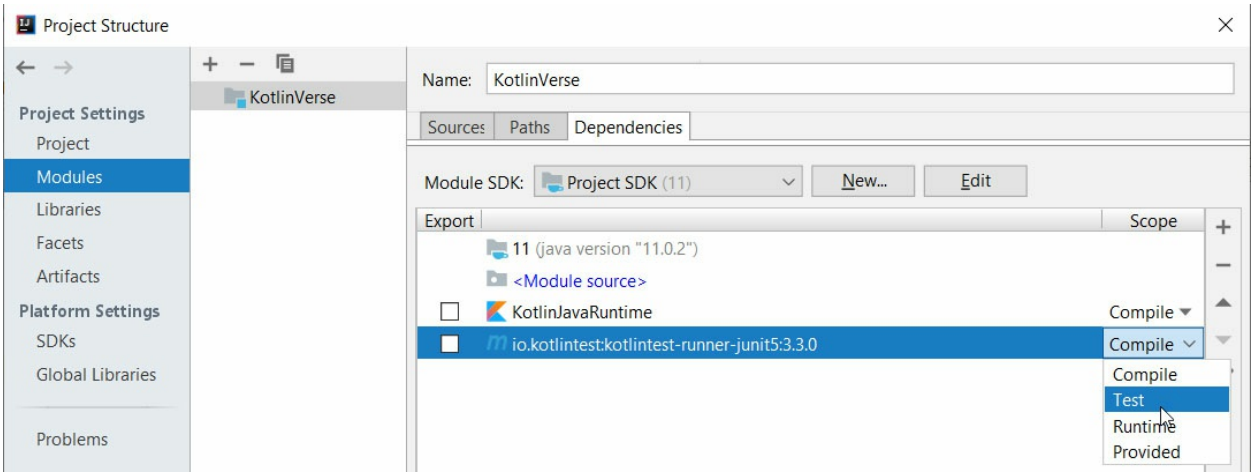

Figure 14.2: *Choosing test dependency scope*

The final preparation step is to configure a directory to contain our test source code. If you don't have it already, create a new directory (say, `test`) right beside `src`, which holds production sources by right clicking on the module root in the **`Project`** view and choosing **`New | Directory`**. Now, we need to tell IDE that it will be our test source root. To do this, right click on the newly added directory and choose **`Mark Directory as | Test Sources Root`**. The test directory will change its color to green showing that IDEA will now treat its content as source files for our tests.

It's also worth installing a special plugin which improves the IntelliJ integration with KotlinTest. You can do it via the **`Plugins`** tab in the Settings dialog (**`File | Settings`**) by searching for `kotlintest` (Figure 14.3). After clicking on **`Install`** and downloading and installing, you'll need to restart the IDE.

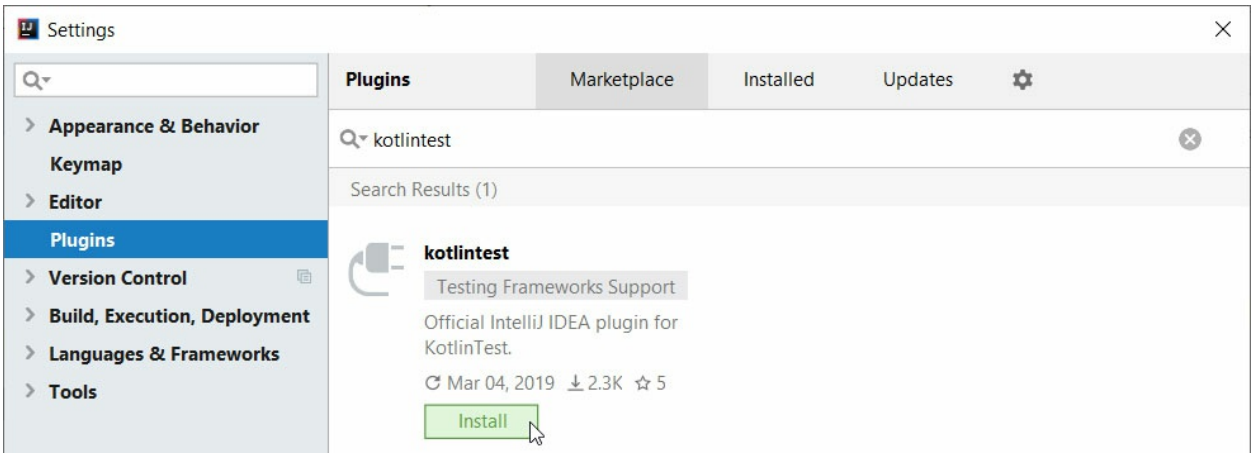

Figure 14.3: *Installing KotlinTest plugin for IntelliJ IDEA*

Now, we can start writing a code just like we did before. Let's create a new file in the `test` directory and write a simple test specification:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.StringSpec

class NumbersTest :StringSpec({
  "2 + 2 should be 4" { (2 + 2) shouldBe 4 }
  "2 * 2 should be 4" { (2 * 2) shouldBe 4 }
})
```

We'll explain the meaning behind this definition in a moment, but even now you'll surely be able to recognize a pair of simple tests named "`2 + 2` should be `4`" and "`2 * 2` should be `4`" after checking some arithmetic identities. To run the test, notice the triangular markers on the left. By clicking one of them, you can execute either corresponding test or a whole specification (*Figure 14.4*):

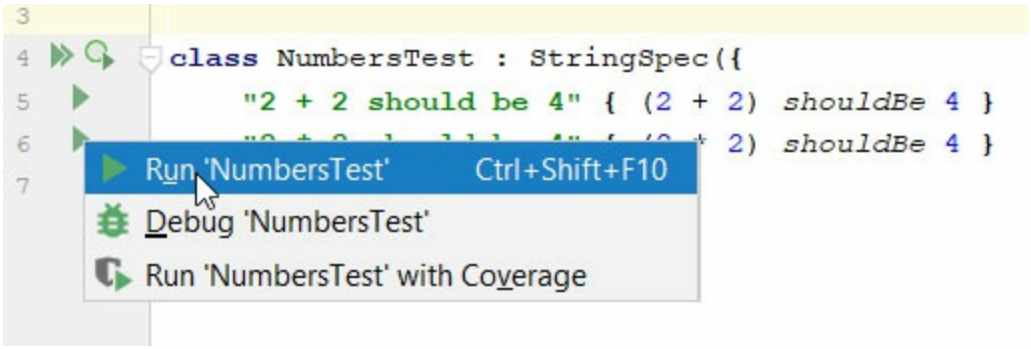

Figure 14.4: *Running KotlinTest specification in IntelliJ*

Now, we're ready to begin our discussion of KotlinTest feature. Our first

topic will be related to various specification styles you can use to organize your test cases.

Specification styles

KotlinTest supports multiple specification styles; each of them affects how your test code is organized. You can easily mix different styles in your project and even define your own by creating implementation of the `AbstractSpec` class or one of its more specific subclasses like `AbstractStringSpec`. In this section, we'll take a look at the styles which are available right out of the box once you add KotlinTest to your project.

To define a test case, you need to inherit from one of the specification classes. Then, you can add tests either in a class initializer, or in a lambda which is passed to the superclass constructor. The way you define tests themselves is style-specific and in most cases, involve some DSL-like API. Let's consider a simple example using the `StringSpec` class:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.StringSpec

class NumbersTest :StringSpec({
  "2 + 2 should be 4" { (2 + 2) shouldBe 4 }
  "2 * 2 should be 4" { (2 * 2) shouldBe 4 }
})
```

With `StringSpec`, individual tests are defined by suspending lambdas placed after a string with a test description. As you might've guessed, this is just an operator form for the `String.invoke()` function defined by `StringSpec`. In this example, the actual verification code uses the shouldBe infix function which throws an exception when its arguments are not equal. This function is a part of matchers DSL which we'll cover in the next section.

Note that `StringSpec` imposes a flat test case structure where all tests in a particular class are defined on the same level. If you try to place one test block inside another, the framework will fail with an exception at runtime.

More complex layout is given by the `WordSpec` class. In the simplest form, it allows you to define two-level hierarchy where tests defined, which are similar to `StringSpec`, are grouped by the calls of the `should()` function:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.WordSpec
```

```
class NumbersTest2 :WordSpec({
 "1 + 2" should {
  "be equal to 3" { (1 + 2) shouldBe 3 }
  "be equal to 2 + 1" { (1 + 2) shouldBe (2 + 1) }
 }
})
```

Additionally, you can define one more level of grouping by wrapping should() calls inside `When()` or `` `when`() ``:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.WordSpec

class NumbersTest2 :WordSpec({
 "Addition" When {
  "1 + 2" should {
   "be equal to 3" { (1 + 2) shouldBe 3 }
   "be equal to 2 + 1" { (1 + 2) shouldBe (2 + 1) }
  }
 }
})
```

What if we want a hierarchy with arbitrary number of levels? The `FunSpec` class wraps the test code inside the `test()` function calls which take a test description and a suspending lambda to run. Unlike `StringSpec`, this style supports grouping of tests by context blocks:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.FunSpec

class NumbersTest :FunSpec({
 test("0 should be equal to 0") { 0 shouldBe 0 }
 context("Arithmetic") {
  context("Addition") {
   test("2 + 2 should be 4") { (2 + 2) shouldBe 4 }
  }
  context("Multiplication") {
   test("2 * 2 should be 4") { (2 * 2) shouldBe 4 }
  }
 }
})
```

Both test and context blocks may be used at any level except inside test block themselves.

IDE Tips: When they run in IDE, the results of such multi-level tests are also presented in a hierarchical view corresponding to the

specification blocks. Figure 14.5 shows the result of running the test code above in IntelliJ IDEA:

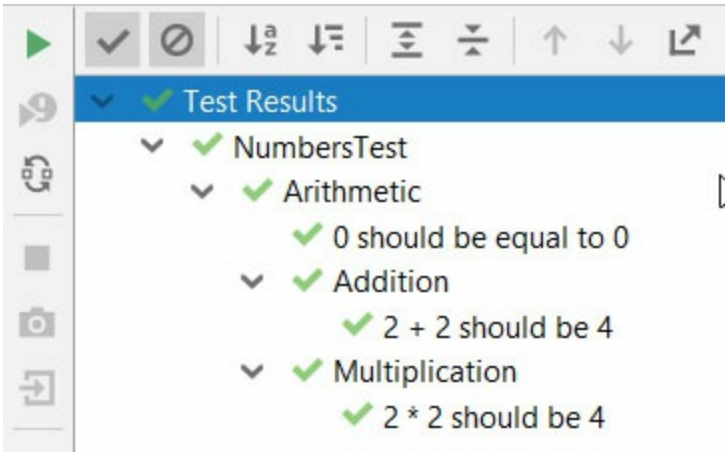

Figure 14.5: *Multi-level test results in IntelliJ IDEA*

The `ExpectSpec` is basically the same but uses `expect()` instead of `test()` and additionally, forbids placing tests at the top level (i.e. all tests must be put within some `context()` block).

The `DescribeSpec` uses the `describe()/context()` blocks for the purpose of grouping with actual tests placed inside `it()`:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.DescribeSpec

class NumbersTest :DescribeSpec({
 describe("Addition") {
  context("1 + 2") {
   it("should give 3") { (1 + 2) shouldBe 3 }
  }
 }
})
```

The `ShouldSpec` produces a layout similar to `FunSpec` with context blocks used for grouping and test blocks placed at the leaf level. The differences are purely syntactic. To define a context block, you will use `invoke()` calls on description string (similarly to test blocks of `StringSpec`), while test blocks themselves are defined by the `should()` function calls:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.ShouldSpec

class NumbersTest :ShouldSpec({
 should("be equal to 0") { 0 shouldBe 0 }
```

```
 "Addition" {
  "1 + 2" {
   should("be equal to 3") { (1 + 2) shouldBe 3 }
   should("be equal to 2 + 1") { (1 + 2) shouldBe (2 + 1) }
  }
 }
})
```

One more specification of a similar kind can be constructed via the `FreeSpec` class. Like `StringSpec`, it uses `invoke()` on strings to define tests while contexts are introduced by the minus operator:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.FreeSpec

class NumbersTest :FreeSpec({
 "0 should be equal to 0" { 0shouldBe 0 }
 "Addition" - {
  "1 + 2" - {
   "1 + 2 should be equal to 3" { (1 + 2) shouldBe 3 }
   "1 + 2 should be equal to 2 + 1" { (1 + 2) shouldBe (2 + 1) }
  }
 }
})
```

KotlinTest also supports BDD (behavior-driven-development) specification styles inspired by the Gherkin language. In the `FeatureSpec`, hierarchy roots are introduced by feature blocks which in turn contain scenario blocks implementing a particular test. The `and()` calls may be used to group scenarios (and other groups) within a particular feature:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.FeatureSpec

class NumbersTest :FeatureSpec({
 feature("Arithmetic") {
  val x = 1
  scenario("x is 1 at first") { x shouldBe 1 }
  and("increasing by") {
   scenario("1 gives 2") { (x + 1) shouldBe 2 }
   scenario("2 gives 3") { (x + 2) shouldBe 3 }
  }
 }
})
```

A similar style is implemented by `BehaviorSpec` which introduces three basic levels denoted by functions `given()/Given()`, `when()/When()`, and

`then()/Then()`. Additional levels of grouping may be introduced by `and()/And()` calls which can combine several `when/then` blocks:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.BehaviorSpec

class NumbersTest :BehaviorSpec({
 Given("Arithmetic") {
  When("x is 1") {
   val x = 1
   And("increased by 1") {
    Then("result is 2") { (x + 1) shouldBe 2 }
   }
  }
 }
})
```

Note how using of these blocks can produce a test description which is very close to natural language ("when x is `1` and increased by `1` then result is `2`").

The final spec style we're going to consider is `AnnotationSpec`. This style doesn't use DSL-like test specification but instead relies on `@Test` annotations you apply to test class methods, which are similar to test frameworks like JUnit or TestNG:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.AnnotationSpec

class NumbersTest :AnnotationSpec() {
 @Test fun `2 + 2 should be 4`() { (2 + 2) shouldBe 4 }
 @Test fun `2 * 2 should be 4`() { (2 * 2) shouldBe 4 }
}
```

You can also disable a particular test by annotating it with `@Ignore`.

Assertions

Matchers

In the previous samples of code demonstrating the use of various specification styles, we've used the `shouldBe` function which asserts simple equality of its arguments. This is just one example of numerous matchers provided by the KotlinTest library.

Matchers are defined as extension functions which can be invoked either in a form of an ordinary call or as infix operator. All matcher names start with

`shouldBe`. This convention facilitates readable names like `shouldBeGreaterThanOrEqual` in the test code. A full list of built-in matchers can be found in the KotlinTest documentation; we won't focus on particular examples here because most of the matcher functions have self-explanatory names and can be used in an intuitive way. In this section, we'll be interested in more advanced issues such as extending testing frameworks with your own matchers.

To define a custom matcher, you need to implement the `Matcher` interface and override its `test()` method:

```
abstract fun test(value: T): Result
```

The `Result` object describes the outcome of matching. It's a data class which contains the following properties:

- `passed`: Checks whether assertion is satisfied (true) or not (false).
- `failureMessage`: A message which is shown when assertion fails and tells what should happen in order to make it pass.
- `negatedFailureMessage`: This is used when you invoke negated version of the matcher and it fails.

Let's, for example, create a matcher which checks whether a given number is odd:

```
import io.kotlintest.Matcher
import io.kotlintest.Result

fun beOdd() = object : Matcher<Int> {
 override fun test(value: Int): Result {
  return Result(
   value % 2 != 0,
   "$value should be odd",
   "$value should not be odd"
  )
 }
}
```

Now, we can use this matcher for our assertions by passing it to a built-in extension function `should()/shouldNot()`:

```
import io.kotlintest.*
import io.kotlintest.specs.StringSpec

class NumberTest :StringSpec({
```

```
  "5 is odd" { 5 should beOdd() }
  "4 is not odd" { 4shouldNotbeOdd() }
})
```

Note that we've chosen the `beOdd` name with an intention to get a human-readable name for the resulting assertion (should be odd/should not be odd).

Any implementation of the `Matcher` interface automatically supports and/or/invert operations which combine matchers following the logic of boolean operations. We can use them to build assertions based on complex predicates such as in the following example which combines `beOdd()` with a built-in `positive()` matcher:

```
"5 is positive odd" { 5 should (beOdd() and positive()) }
```

One more operation supported by matchers is `combine()` which allows you to generalize the existing matcher to a new type by providing the conversion function. The following function reuses the `beOdd()` matcher to assert that a given collection has odd length:

```
fun beOddLength() = beOdd().compose<Collection<*>> { it.size }
```

Note that while all matchers can be called via the `should()/shouldNot()` function, many built-in ones are also accompanied by specialized functions whose names start with should. For example, the following assertions are equivalent:

```
5 should beLessThan(10)
5 shouldBeLessThan(10)
```

Inspectors

Apart from matchers, KotlinTest also supports a related concept of inspectors. Inspector is an extension function for some collection class which allows you to verify that a given assertion holds for some group of its elements:

- `forAll()/forNone()`: Checks whether all/none elements satisfy an assertion.
- `forExactly(n)`: Check whether exactly n elements satisfy an assertion; there is also a special `forOne()` function which handles the case `n = 1`.
- `forAtLeast(n)/forAtMost(n)`: Check whether at least/at most n

elements satisfy an assertion; when `n = 1` you can also use specialized inspectors `forAtLeastOne()/forAtMostOne()` or `forAny()` which is the same as `forAtLeastOne()`.

- `forSome()`: Check whether some but not all elements satisfy an assertion.

Let's consider an example of using these inspectors:

```
import io.kotlintest.inspectors.*
import io.kotlintest.matchers.numerics.shouldBeGreaterThanOrEqual
import io.kotlintest.shouldBe
import io.kotlintest.specs.StringSpec

class NumberTest :StringSpec({
 val numbers = Array(10) { it + 1 }

 "all are non-negative" {
  numbers.forAll { it shouldBeGreaterThanOrEqual 0 }
 }
 "none is zero" { numbers.forNone { it shouldBe 0 } }
 "a single 10" { numbers.forOne { it shouldBe 10 } }
 "at most one 0" { numbers.forAtMostOne { it shouldBe 0 } }
 "at least one odd number" {
  numbers.forAtLeastOne { it % 2 shouldBe 1 }
 }
 "at most five odd numbers" {
  numbers.forAtMost(5) { it % 2 shouldBe 1 }
 }
 "at least three even numbers" {
  numbers.forAtLeast(3) { it % 2 shouldBe 0 }
 }
 "some numbers are odd" { numbers.forAny { it % 2 shouldBe 1 } }
 "some but not all numbers are even" {
  numbers.forSome { it % 2 shouldBe 0 }
 }
 "exactly five numbers are even" {
  numbers.forExactly(5) { it % 2 shouldBe 0 }
 }
})
```

Handling exceptions

KotlinTest has a special `shouldThrow()` assertion which checks whether some code fails due to a specific exception. This is a convenient alternative of catching exceptions with an explicit `try/catch` block. On being successful,

`shouldThrow()` returns caught exception which you can inspect afterwards:

```
import io.kotlintest.matchers.string.shouldEndWith
import io.kotlintest.shouldThrow
import io.kotlintest.specs.StringSpec

class ParseTest :StringSpec({
 "invalid string" {
  val e = shouldThrow<NumberFormatException>{ "abc".toInt() }
  e.messageshouldEndWith "\"abc\""
 }
})
```

A useful exception-related feature of the KotlinTest is its ability to temporarily suppress exceptions thrown by failed assertions. This is called soft assertion and may be useful if your test consists of several assertions and you want to see all that failed. Normally, this doesn't happen because tests terminate after the first thrown exception. KotlinTest allows you to work around this behavior using `assertSoftly` blocks. `AssertionError` exceptions are automatically caught inside the block and accumulated letting all assertions to run (unless they fail with some other exception). When the block is finished, `assertSoftly` packs all accumulated exceptions (if there are any) into a single `AssertionError` and throws it back to the caller. Let's consider an example:

```
import io.kotlintest.assertSoftly
import io.kotlintest.inspectors.forAll
import io.kotlintest.specs.StringSpec

class NumberTest :StringSpec({
 val numbers = Array(10) { it + 1 }

 "invalid numbers" {
 assertSoftly {
    numbers.forAll { it shouldBeLessThan 5 }
    numbers.forAll { it shouldBeGreaterThan 3 }
  }
 }
})
```

Without `assertSoftly()`, the second `forAll()` assertion won't even be checked after the first one fails. Now, both assertions are executed and the test fails with an exception:

```
io.kotlintest.tables.MultiAssertionError:
The following 9 assertions failed
```

…

As you can the resulting message lists all individual failures.

Testing non-deterministic code

When having to deal with a non-deterministic code, which sometimes passes only after several attempts, you can use a convenient alternative to timeouts and multiple invocations. The `eventually()` function verifies that the given assertion is satisfied at least once within a specified period of time:

```
import io.kotlintest.*
import io.kotlintest.specs.StringSpec
import java.io.File

class StringSpecWithConfig :StringSpec({
 eventually(10.seconds) {
   // Check that file eventually contains a single line
   // (within 10 seconds)
   File("data.txt").readLines().size shouldBe 1
 }
})
```

The `continually()` function similarly verifies that nested assertions are satisfied upon its call and remains within a specified interval:

```
import io.kotlintest.*
import io.kotlintest.specs.StringSpec
import java.io.File

class StringSpecWithConfig :StringSpec({
 // Check that file contains a single line
 // and line count doesn't change for at least 10 seconds
 continually(10.seconds) {
   File("data.txt").readLines().size shouldBe 1
 }
})
```

Property-based testing

KotlinTest is capable of property-based testing where you specify some predicates and have KotlinTest automatically generate a random test data for verify it against. This technique is useful when we want to check whether some condition holds over a large set of values which is hard to prepare and maintain manually.

For example, suppose that we define a function which computes the minimum of two numbers:

```
infix fun Int.min(n: Int) = if (this < n) this else n
```

You want to ensure that its result is always less or equal to each argument. To do this, we wrap the respective assertion inside the `assertAll()` call:

```
import io.kotlintest.matchers.beLessThanOrEqualTo
import io.kotlintest.properties.assertAll
import io.kotlintest.should
import io.kotlintest.specs.StringSpec

class NumbersTest: StringSpec({
 "min" {
  assertAll{ a: Int, b: Int ->
   (a min b).let {
    it should (beLessThanOrEqualTo(a) and
    beLessThanOrEqualTo(b))
   }
  }
 }
})
```

When you run this code, KotlinTest will generate a stream of Int pairs and test all of them against our assertion. By default, the test dataset consists of 1000 items, but you can specify its size explicitly as an argument of `assertAll()`.

There is also `theassertNone()` function which checks whether none of the generated items satisfy given assertion.

As an alternative, we can use the `forAll()/forNone()` function which take a lambda with the Boolean return type and verifies that all/none generated items have corresponding predicates that evaluate to true:

```
import io.kotlintest.properties.forAll
import io.kotlintest.specs.StringSpec

class NumbersTest: StringSpec({
 "min" {
  forAll{ a: Int, b: Int ->
   (a min b).let { it <= a && it <= b }
  }
 }
})
```

KotlinTest have default generators for many common types such as `Int`, `Boolean` and `String`. By default, it uses runtime information of lambda parameter types to choose the generator automatically. Sometimes, though, this may be unwanted or even not possible if a type in question is simply not supported. In this case, we need to specify the generator explicitly as an instance of the Gen interface. Its companion object contains a bunch of helpful methods you can use to construct various implementations of generators. In particular (the rest can be as usually found in the documentation):

- `choose(min, max)`: Generate random integers in the range from min to max (excluding max).
- `positiveIntegers()/negativeIntegers()/nats()`: Generate stream of random positive/negative/non-negative integers.
- `from(collection)`: Take random elements of a given list or array.

You may also define your own generator. One way to do it is to use the Gen.`create()` function which builds a generator based on a specified lambda. In *Chapter 11, Domain-Specific Languages*, we've defined a class for representation of rational numbers. Let's now check whether subtracting any rational from itself produces a zero. To do this, we need to implement a custom generator:

```
import io.kotlintest.properties.*
import io.kotlintest.specs.StringSpec
import kotlin.random.Random

class NumbersTest: StringSpec({
 "Subtraction" {
  forAll(genRationals()) { a: Rational ->
   (a - a).num == 0
  }
 }
}) {
 companion object {
  private fun genRationals(): Gen<Rational> {
   return Gen.create {
    val num = Random.nextInt()
    val den = Random.nextInt()
    Rational.of(num, if (den != 0) den else 1)
   }
  }
 }
```

```
}
```

An alternative way is to inherit from the `Gen<T>` interface directly. In this case, you need to provide an implementation of two methods:

- `constants()`: Returns a collection of `T` value which are always included into the generated stream; these values are meant for various corner cases (say, default generator for integers uses `Int.MIN_VALUE, 0` and `Int.MAX_VALUE` for this purpose).
- `random()`: Returns a sequence of random elements of type `T`.

For example, we can rewrite our `Rational` generator to the following object:

```
object RationalGen : Gen<Rational> {
 override fun constants(): Iterable<Rational> {
  return listOf(Rational.of(0), Rational.of(1), Rational.of(-1))
 }

 override fun random(): Sequence<Rational> {
  return generateSequence {
   val num = Random.nextInt()
   val den = Random.nextInt()
   Rational.of(num, if (den != 0) den else 1)
  }
 }
}
```

It’s also possible to use a fixed test data set rather than using random values provided by the framework or custom generator. For this, you need to use the `forall()` function which takes a `vararg` of row objects:

```
import io.kotlintest.data.forall
import io.kotlintest.specs.StringSpec
import io.kotlintest.tables.row

class NumbersTest: StringSpec({
 "Minimum" {
  forall(
   row(1, 1),
   row(1, 2),
   row(2, 1)
  ) { a: Int, b: Int ->
   (a min b).let { it <= a && it <= b }
  }
 }
})
```

Note the difference in naming: `forall` vs. `forAll`.

On top of this, you can pack multiple rows into a single table object which also has a specific set of headers. The headers are then used to provide a context information when the test fails. For example:

```
import io.kotlintest.matchers.numerics.shouldBeGreaterThanOrEqual
import io.kotlintest.specs.StringSpec
import io.kotlintest.tables.*

class NumbersTest :StringSpec({
 "Minimum" {
  forAll(
   table(
    headers("name", "age"),
    row("John", 20),
    row("Harry", 25),
    row("Bob", 16)
   )
  ) { name, age ->
   age shouldBeGreaterThanOrEqual 18
  }
 }
})
```

Running the preceding test will produce an error with the following message:

```
Test failed for (name, Bob), (age, 16) with error 16 should be >=
18
```

This option is supported by both `forAll()` and `forNone()` overloads. Note, however, that unlike `forAll()/forNone()` used for generator-based testing, these functions take a lambda with the Unit return type. That's why we've used matcher functions instead of simply returning a Boolean value.

Fixtures and configurations

Providing a fixture

It's often the case that tests need some kind of code which initializes the necessary environment and resources (also known as test fixture) before actual test invocation and finalizes them afterwards. In KotlinTest, you can use the `TestListener` interface to embed your code into various stages of test case lifecycle. Let's see what methods it has:

- `beforeProject()/afterProject()`: Invoked upon test engine start/finish.
- `beforeSpecClass()`: Invoked once (regardless of how many times a given specification class is instantiated) before any tests corresponding to a given specification class are started. `afterSpecClass()`: Invoked after all such tests are finished.
- `beforeSpec()`: Invoked after instantiating a specification but before running its tests; `afterSpec()` is invoked after all tests for a given specification instance are finished.
- `beforeTest()/afterTest()`: Invoked before/after running a particular test block.

In order to have some effect, a listener instance must be registered in a particular specification class by overriding its `listener()` method:

```
import io.kotlintest.*
import io.kotlintest.extensions.*
import io.kotlintest.specs.FunSpec

object MyListener :TestListener {
 override fun beforeSpecClass(spec: Spec, tests:
 List<TopLevelTest>) {
  println("Before spec class: ${spec.description()}")
 }

 override fun beforeSpec(spec: Spec) {
  println("Before spec: ${spec.description()}")
 }

 override fun beforeTest(testCase: TestCase) {
  println("Before test: ${testCase.name}")
 }

 override fun afterTest(testCase: TestCase, result: TestResult)
 {
  println("After test: ${testCase.name}")
 }

 override fun afterSpec(spec: Spec) {
  println("After spec: ${spec.description()}")
 }

 override fun afterSpecClass(spec: Spec,
  results: Map<TestCase, TestResult>) {
  println("After spec class: ${spec.description()}")
```

```
 }
}

class NumbersTest :FunSpec() {
 init {
  context("Increment") {
   test("2+2") {
    2 + 2 shouldBe 4
   }
   test("2 * 2") {
    2 * 2 shouldBe 4
   }
  }
 }

 override fun listeners() = listOf(MyListener)
}
```

Running the preceding code will produce the following output:

```
Before spec class: Description(parents=[], name=NamesTest)
Before spec: Description(parents=[], name=NamesTest)
Before test: IncrementBefore test: 2+2
After test: 2+2
Before test: 2 * 2
After test: 2 * 2
After test: Increment
After spec: Description(parents=[], name=NamesTest)
After spec class: Description(parents=[], name=NamesTest)
```

Note that in our example, beforeSpec()/afterSpec() are invoked only once just like beforeSpecClass()/afterSpecClass() because only one instance ofNumbersTest is created. This is not always the case as you can configure the framework to create a new specification per each test (see isolation mode discussion in the Test configuration section).

The key difference between `beforeSpec()` and `beforeTest()`(as well as between `afterSpec()` and `afterTest())` is that `beforeSpec()` is invoked only if the test in question is enabled. In the following section, we'll see how you can switch off individual tests using configurations.

If you want to provide an implementation of the `beforeProject()/afterProject()` methods, you need to register a global listener using the `ProjectConfig` singleton. This singleton must inherit from the `AbstractProjectConfig` class and be placed in the `io.kotlintest.provided` package:

```
package io.kotlintest.provided

import io.kotlintest.*
import io.kotlintest.extensions.*

object ProjectListener :TestListener {
 override fun beforeProject() { println("Before project") }
 override fun afterProject() { println("After project") }
}

object ProjectConfig :AbstractProjectConfig() {
 override fun listeners(): List<TestListener> {
  return listOf(ProjectListener)
 }
}
```

One more useful feature of KotlinTest is its ability to automatically close resources which implement the `AutoCloseable` interface. For this to work, you need to register a resource upon its allocation with the `autoClose()` call:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.FunSpec
import java.io.FileReader

class FileTest :FunSpec() {
 val reader = autoClose(FileReader("data.txt"))

 init {
  test("Line count") {
   reader.readLines().isNotEmpty() shouldBe true
  }
 }
}
```

Test configuration

KotlinTest gives you a set of options to configure a testing environment. In particular, specification classes provide the `config()` function which can be used to set various test execution parameters. Its usage depends on a chosen specification style but in general, it replaced an ordinary test block. Let's consider some examples:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.*
import java.time.Duration

class StringSpecWithConfig :StringSpec({
```

```
  "2 + 2 should be 4".config(invocations = 10) { (2 + 2) shouldBe
  4 }
})

class ShouldSpecWithConfig :ShouldSpec({
  "Addition" {
    "1 + 2" {
      should("be equal to 3").config(threads = 2, invocations =
      100) {
        (1 + 2) shouldBe 3
      }
      should("be equal to 2 + 1").config(timeout = 1.minutes) {
        (1 + 2) shouldBe (2 + 1)
      }
    }
  }
})

class BehaviorSpecWithConfig :BehaviorSpec({
  Given("Arithmetic") {
    When("x is 1") {
      val x = 1
      And("increased by 1") {
        then("result is 2").config(invocations = 100) {
          (x + 1) shouldBe 2
        }
      }
    }
  }
})
```

You can find more detailed information in the documentation on a particular specification style.

Let’s see what parameters we can control using the `config()` function:

- `invocations`: Number of times to execute a test; a test is considered passed only if all invocations succeed. This option may be useful for non-deterministic tests which may fail only occasionally.
- `threads`: Number of threads to use when running a test. This parameter is only meaningful when invocations is greater than 1 since otherwise there is nothing to parallelize.
- `enabled`: Whether a test should be run; setting to false disables test execution.
- `timeout`: A duration object representing maximum time for test to run.

If the test execution time exceeds this timeout, it's terminated and considered failed. Like invocation count, this option is useful for non-deterministic tests.

Note that the threads option affects only parallelizing of individual tests within a test case. If you want to run multiple test cases in parallel as well, you need to use the `AbstractProjectConfig` which we've discussed earlier. Just override its `parallelism()` method and return a desired number of concurrent threads:

```
package io.kotlin.provided

import io.kotlintest.AbstractProjectConfig

object ProjectConfig :AbstractProjectConfig() {
 override fun parallelism(): Int = 4
}
```

Apart of configuring each test individually, you may also specify a common configuration for all tests of a particular test case by overriding the `defaultTestCaseConfig` property:

```
import io.kotlintest.TestCaseConfig
import io.kotlintest.shouldBe
import io.kotlintest.specs.StringSpec

class StringSpecWithConfig :StringSpec({
 "2 + 2 should be 4" { (2 + 2) shouldBe 4 }
}) {
 override valdefaultTestCaseConfig: TestCaseConfig
 get() = TestCaseConfig(invocations = 10, threads = 2)
}
```

Default configuration options are inherited by tests unless you specify their own configuration explicitly.

One more feature of KotlinTest we'd like to point out in conclusion is an ability to choose how a test case instance is shared between its tests. This is so called an isolation mode. By default, the test case is instantiated only once and its instance is used to run all its tests. Although this is good from the performance point of view, there are some scenarios when such a policy is undesired; if the test case has a mutable state which is read and modified by individual tests. In such cases, you may want to instantiate the test each time you start a test or test group. To achieve this, you just need to override the `isolationMode()` method of your test case class. This method returns a value

of IsolationModeenum which defines three options:

- `SingleInstance`: A single instance of the test case is created; this is the default behavior.
- `InstancePerTest`: A new instance of the test case is created each time a context or test block is executed.
- `InstancePerLeaf`: The test is instantiated before executing an individual test block.

Let's consider an example. Suppose we have the following `FunSpec`-style test case:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.FunSpec

class IncTest :FunSpec() {
 var x = 0

 init {
  context("Increment") {
   println("Increment")
   test("prefix") {
    println(“prefix)
    ++x shouldBe 1
   }
   test("postfix") {
    println(“postfix)
    x++ shouldBe 0
   }
  }
 }
}
```

If you run it, you'll see that the second test fails. This happens because the x variable retains a value assigned in the prefix block. If we change isolation mode to `InstancePerTest` by adding:

```
override fun isolationMode() = IsolationMode.InstancePerTest
```

Both tests will pass since each of them will get its own `IncTest` instance. The messages printed to the standard output will look like the following:

```
Running context
Running context
prefix
Running context
```

```
postfix
```

This happens because `IncTest` is instantiated three times. First time to execute the context block itself, second to execute the prefix test which also runs context block the second time and third for the postfix text (which again requires to run context block first). As a result, the context block is also executed three times.

If we change isolation mode to `InstancePerLeaf`, the context blocks won't be executed by themselves but only as a part of running individual tests. As a result, `IncTest` will be instantiated only two times (once for prefix and once for postfix) and the output will look like the following:

```
Running context
prefix
Running context
postfix
```

This concludes our overview of the KotlinTest framework. For more detailed information about the features we've mentioned (as well as those we haven't), the reader is advised to follow documentation available at

https://github.com/kotlintest/kotlintest.

Conclusion

In this chapter, we learned about the basics of writing test specifications in KotlinTest, a popular open-source testing framework designed specifically for Kotlin-powered applications. We discussed how to organize our test code using out-of-the-box specification style, how to write expressive and easy-to-read tests with matchers and inspectors, how to describe test data sets and make use of automatic property testing. We also explained the use of KotlinTest set up/tear down facilities and basic test configurations. Now, you have all necessary knowledge to write your own test specifications and to learn more advanced features of KotlinTest and other testing frameworks.

In the next chapter, we'll talk about using Kotlin to build applications for the Android platform. We'll explain how to configure a project in Android Studio, discuss basic UI and activity lifecycle, and introduce you to various useful features provided by Android extensions and Anko frameworks.

Questions

1. Give an overview of popular testing frameworks with Kotlin support.
2. Describe specification styles supported by KotlinTest.
3. What is a matcher? How can you combine and transform matchers for writing complex assertions?
4. Explain how to implement custom KotlinTest matcher.
5. Describe the `shouldThrow()` function. What is a soft assertion?
6. Describe collection inspectors available in KotlinTest.
7. Explain the meaning of `eventually()` and `continually()` functions.
8. How can you implement initialization and finalization of test resources using listeners?
9. How to specify a configuration for individual tests and specifications? How to define a global configuration?
10. Explain the differences between test case isolation modes.

CHAPTER 15

Android Applications

In this chapter, we'll talk about using Kotlin in the development of applications targeting the Android platform. Thanks to the excellent programming experience given by the language and official support from the Google, this niche of Kotlin has become one of the most flourishing in its entire ecosystem. Comprehensive discussion of the Android platform fundamentals, however, does beyond a doubt deserve a separate book, so our task here will be quite modest to serve as a kind of introduction to the Android world, provoking further investigation and learning.

The chapter is divided into two parts. In the first part, we'll talk about basic features of Android Studio, how to set up a new project, how Gradle is used for project configuration and build, what is an Android activity, and how you can run applications using an Android device emulator. In the second part, we will focus on the development of a sample calculator application and discuss more advanced issues such as using Kotlin Android Extensions and AnkoLayouts and preserving the activity state.

Structure

- Getting started with Android
- Activities

Objective

Get a basic understanding of using Android Studio and Kotlin for development of applications for the Android platform.

Getting started with Android

In this section, we'll introduce you to the Android Studio IDE and demonstrate a basic project structure on the example of a simple "Hello,

World”-like application.

Setting up an Android Studio project

We’ll start with the basic steps required to configure a project in Android Studio, the official Android IDE developed by Google. Android Studio is based on the JetBrains IntelliJ platform and thus is very similar to the IntelliJ IDEA we’ve referred to in the previous chapters. Unlike IDEA itself, it comes with a set of features specifically targeting at the development of Android applications. Like IntelliJ IDEA, Android Studio has out-of-the-box support of the Kotlin language.

If you don’t have an Android Studio installed yet, you can download the latest version from **https://developer.android.com/studio** and follow installer instructions at **https://developer.android.com/studio/install.html.** In this chapter, we’ll be using Android Studio 3.4.2.

After starting, Android Studio will present you a welcome screen where you can click on a **`Start a new Android Studio project`** link to start a project wizard. If you’ve already opened a project before, Android Studio won’t show you a welcome screen and load that project instead. In this case, you can use the **`File | New | New Project…`** command from the IDE menu.

The first step of the wizard will ask you to choose a template for the first activity in your project (*Figure 15.1*). An activity is basically a representation of a single thing a user will be able to do using your application such as editing notes, showing current time, or in the case of our first application, simply presenting a welcome message to the user. Let’s choose **`Empty Activity`** for now to get Android Studio generate a stub activity for us and click on **`Next`**.

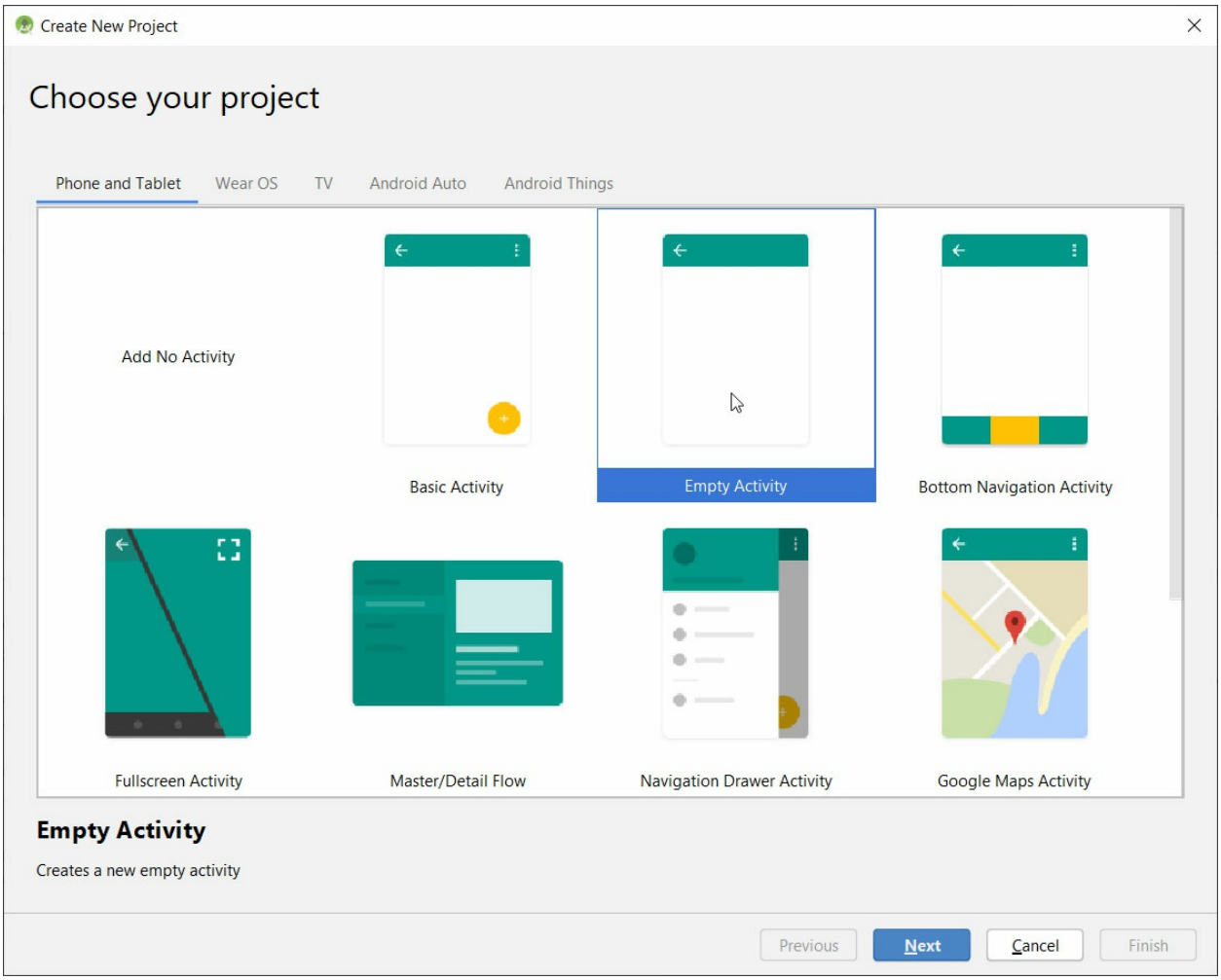

Figure 15.1: *Choosing activity for a new Android project*

The next step of the project wizard requires you to enter basic project information such as its name, common package for project classes, root directory, and default language (see Figure 15.2 for example). You may also choose a minimal version of Android SDK your application will support. The higher the version, the more powerful API you get at your disposal, but at the same time, the less fraction of devices your application can run on. You can see a comparison chart of different API versions by clicking on the **`Help me choose`** link. For our simple example, we can just leave an option suggested by default.

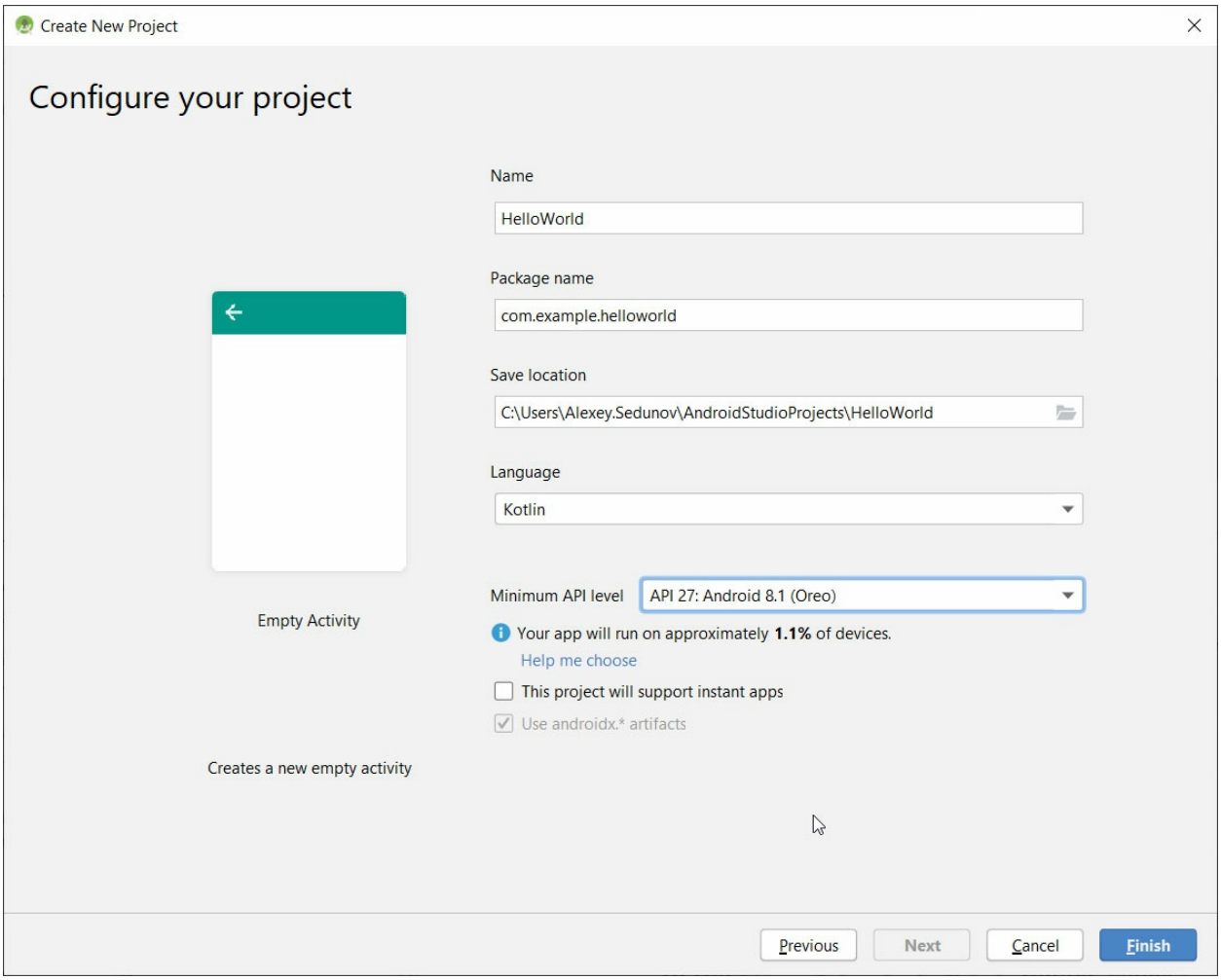

Figure 15.2: *Choosing basic project configuration*

After you click on **`Finish`**, Android Studio will automatically generate necessary project files, including an activity class, a basic set of application resources such as UI layout and configuration files. It will then proceed with configuring a new project (notice the **`Building Gradle project info`** progress bar).

Figure 15.3 demonstrates a structure of a new project as shown by the **`IDE Project`** view. Note that by default this view is presented in the Android mode which groups project source files by corresponding modules and source roots (such as ordinary sources files, generated files and resources).

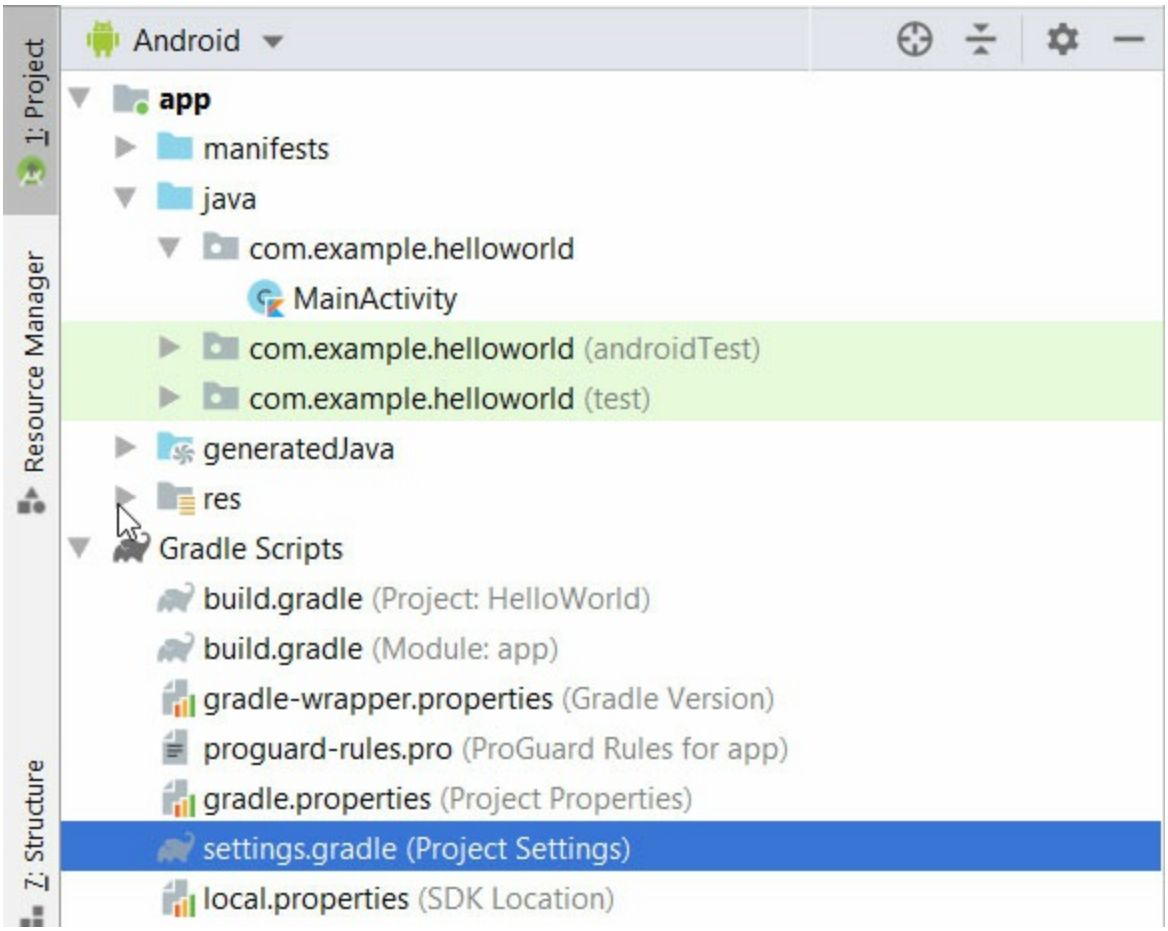

Figure 15.3: *A sample project structure*

A separate node in the Android view, **`Gradle Scripts`** contains files describing the project configuration. Let's look at them in more details.

Gradle build scripts

Android Studio relies on Gradle, a popular build system which automates tasks such as managing project dependencies, compilation, testing, and packaging. Project configuration is described in `build.gradle` files using a domain-specific language and is written in Groovy which is reminiscent of both the Java and Kotlin code (in fact, Groovy is one of the languages that had inspired the Kotlin design). We won't delve into the details of Groovy or Gradle here and instead just highlight some important points related to using Kotlin in Android applications. The project template used by the Android Studio wizard automatically generates the following files:

- The project-level `build.gradle` located in the project root directory.

This file contains common configuration of the entire project.

- `settings.gradle` (also in the project root) specifies which modules are included in the project and may optionally contain some additional configuration commands.
- `local.properties` and `gradle.properties` contain a set of key-value pairs which define properties used in the Gradle scripts such as the path to the Android SDK directory or JVM arguments passed when starting the Gradle process.
- Module-level `build.gradle` which is located in the root directory of the app module contains module-specific configuration.

Let's take a look at the root `build.gradle` file:

```
buildscript {
 ext.kotlin_version = '1.3.31'
 repositories {
  google()
  jcenter()
 }
 dependencies {
  classpath 'com.android.tools.build:gradle:3.4.2'
  classpath "org.jetbrains.kotlin:kotlin-gradle-
  plugin:$kotlin_version"
  // NOTE: Do not place your application dependencies here; they
  belong
  // in the individual module build.gradle files
 }
}
allprojects {
 repositories {
  google()
  jcenter()
 }
}
task clean(type: Delete) {
 delete rootProject.buildDir
}
```

This script basically does the following:

- Defines the `kotlin_version` property which contains a version of the `kotlin` standard library and can be referred to by other scripts.
- Tells Gradle to use `com.android.tools.build:gradle` and

`org.jetbrains.kotlin:kotlin-gradle-plugin` plugins when building projects. This first one adds support of Android modules while the second allows you to build projects with the Kotlin source code.

- Configures a default list of repositories to download dependency artifacts such as binaries and sources of libraries.
- Adds the `clean` task which is used to delete previous compilation results before the project rebuild.

The `settings.gradle` file is quite simple and by default, contains a single include command which tells Gradle which modules make up the out project:

```
include ':app'
```

Let's now look at `build.gradle` which defines the configuration of our Android module:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
 compileSdkVersion 29
 buildToolsVersion "29.0.2"
 defaultConfig {
  applicationId "com.example.helloworld"
  minSdkVersion 27
  targetSdkVersion 29
  versionCode 1
  versionName "1.0"
  testInstrumentationRunner
  "androidx.test.runner.AndroidJUnitRunner"
 }
 buildTypes {
  release {
  minifyEnabled false
  proguardFiles
  getDefaultProguardFile('proguard-android-optimize.txt'),
  'proguard-rules.pro'
  }
 }
}

dependencies {
 implementation fileTree(dir: 'libs', include: ['*.jar'])
 implementation
"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
```

```
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation
    'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-
    core:3.2.0'
}
```

The first thing it does is to enable Android and Kotlin-specific plugins which are added to the root build file. Note that there are two Kotlin plugins here. `kotlin-android` adds basic support for building Android applications with Kotlin sources and `kotlin-android-extensions` enables a special Kotlin compiler extension which simplifies access to UI resources. We'll see an example of how to do it in the upcoming section.

The Android block contains various Android-specific configuration parameters such as application ID, version number, minimal supported version of Android SDK, and so on.

Finally, the dependencies block lists all external dependencies of our module. Each dependency has a definite configuration which is specified first and followed by the dependency description (usually in the form of Maven coordinates like `androidx.core:core-ktx:1.0.2`). A configuration determines when and where this dependency is used; for example, implementation dependencies are added to the compilation classpath and packaged to the build output, but not available during compilation of dependent modules, while `testImplementation` dependencies are added to the compilation classpath of modules tests and used during test execution. As shown in the preceding code, the Kotlin standard library gets automatically added to new module dependencies.

Activity

Let's now look at source files under the java root. This directory can contain both Java and Kotlin files. Find `MainActivity.kt` and open it in the editor window. You'll see something like the following code:

```
package com.example.helloworld

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
```

```
class MainActivity :AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)
 }
}
```

This is an activity class which Android Studio has generated based on the template you've chosen in the project wizard. All activity classes are derived from the `Activity` class which is a part of the Android SDK. The generated class inherits from a more specific `AppCompatActivity` which adds support of a toolbar where you may show the application name and various interactive UI components.

The `onCreate()` method is invoked by the Android OS upon creating an activity instance, so it's a common place for an initialization code. In particular, this method sets up an activity view:

```
setContentView(R.layout.activity_main)
```

The `R` class is automatically generated during compilation of the Android project. It contains identifiers of all resources put in the `res` directory. `R.layout.activity_main`, in particular, corresponds to `activity_main.xml` file in the res/layout. This is called layout XML file which contains description of UI components making up an activity view. If you open it in the editor, Android Studio will present you a UI designer tool you can use to edit UI by dragging and dropping components and changing their properties in the Attributes window. For example, let's choose a `TextView` in the center and change its text size to `36`. You can see the result in Figure 15.4:

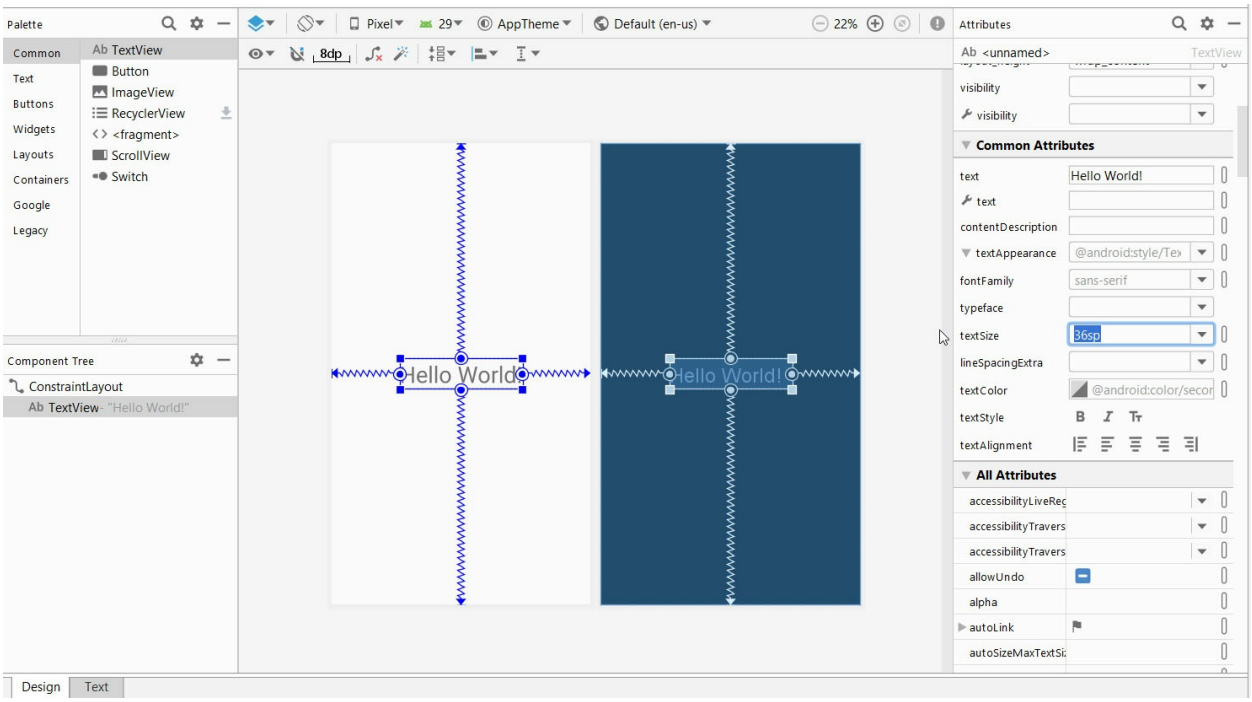

Figure 15.4: *UI Designer*

If you click on the **`Text`** tab at the bottom of the designer window, the editor will turn to a textual view where you can edit the layout like any other XML file. After changing the text size, the XML file will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity">
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Hello World!"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintLeft_toLeftOf="parent"
app:layout_constraintRight_toRightOf="parent"
app:layout_constraintTop_toTopOf="parent"
android:textSize="36sp"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

In the following sections, we'll see an example of more complex UI layout of a calculator application. For now, let's see how this "Hello, World" sample

might look like on an Android device.

Using an Emulator

Now, we can try to run our simple application using an Android device emulator. If you haven't used an emulator yet, the first step is to configure a virtual device. To do this, choose the **`Tools | AVD Manager`** command from the Android Studio menu and click on the **`Create Virtual Device…`** button in the **`Android Virtual Device Manager`** dialog box.

In the dialog box that follows (see Figure 15.5), you can choose a phone model. For our example, the default choice should be **`OK`**, so you can just click on **`Next`**.

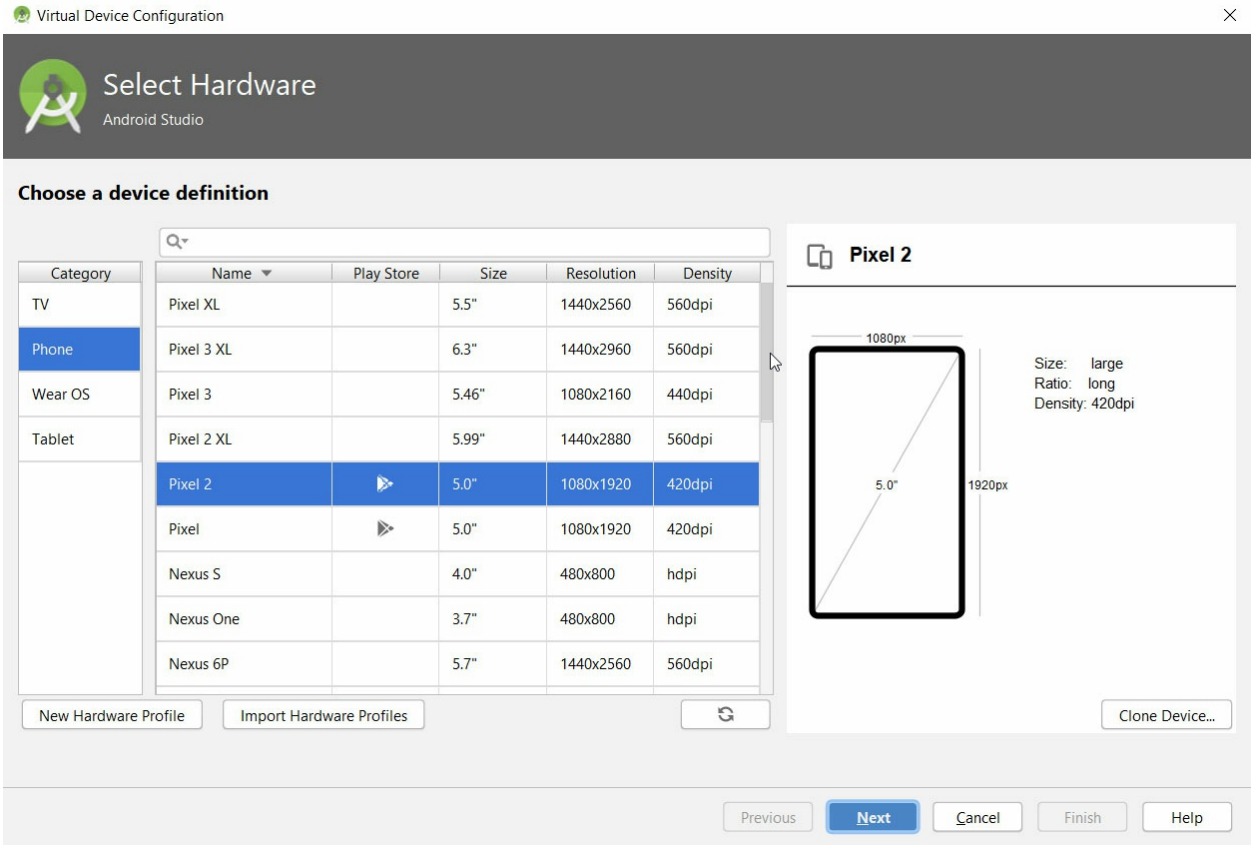

Figure 15.5: *Choosing virtual device configuration*

The **`System Image`** dialog allows you to choose an Android OS image to be used with an emulator. You need to download a chosen image before it is used first by clicking on the corresponding **`Download`** link (Figure 15.6). When the download completes, click on **`Next`**.

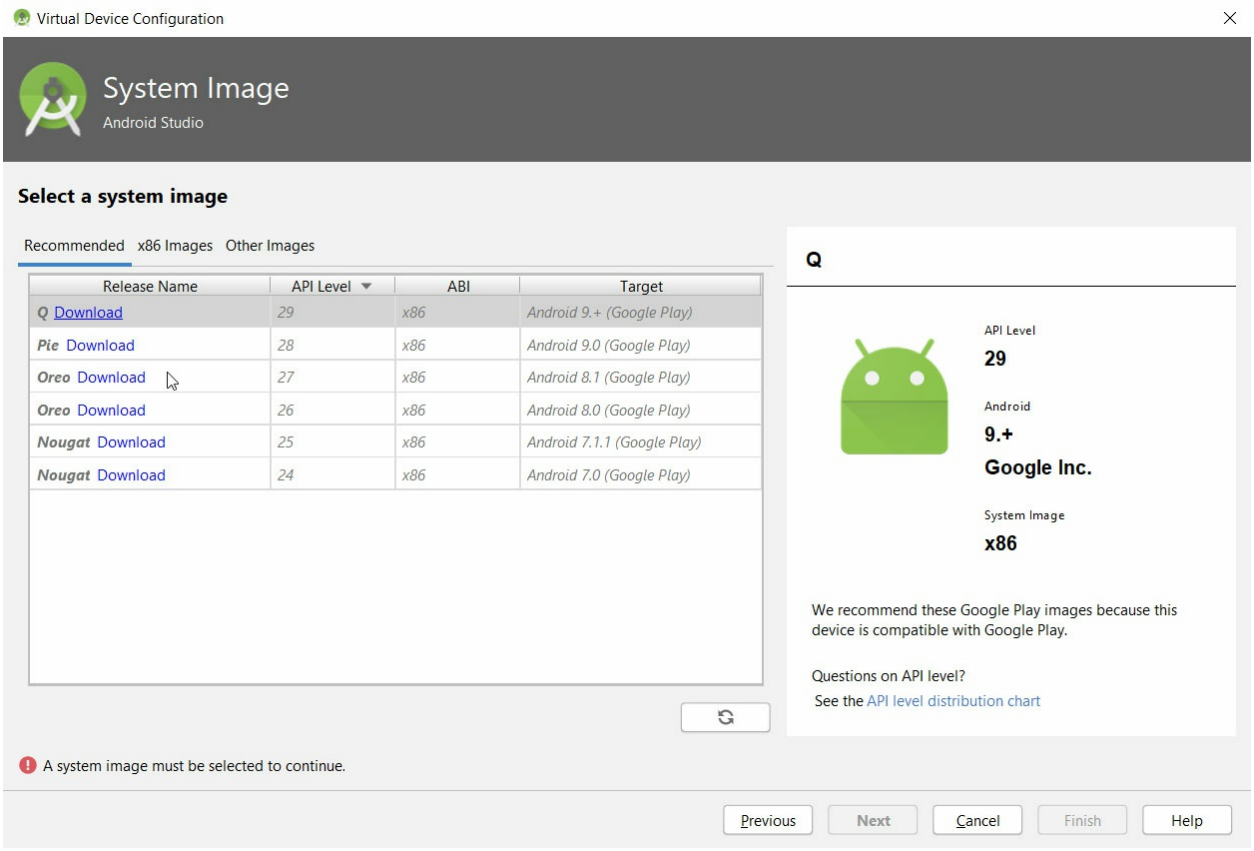

Figure 15.6: *Choosing Android OS image for a virtual device*

In the final **`Verify Configuration`** dialog of the virtual device configuration wizard, you can specify a new device name and choose its default orientation. Clicking on **`Finish`** will bring you back to virtual device manager. Notice the newly added device in the list and close the AVD window.

Let's use our new emulator to run the main activity. To do this, choose the **`Run | Run 'app'`** command or click on the **`Run`** tool button (see Figure 15.7 for example):

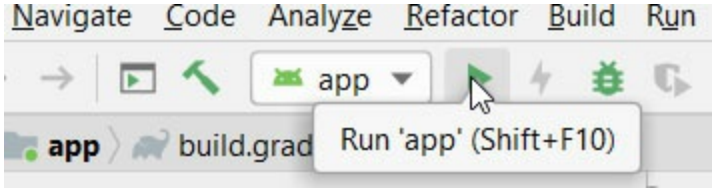

Figure 15.7: *Using Run command*

The Android Studio will bring up a dialog box to choose a virtual device. Choose the one you've just configured and click on **`OK`**. The IDE will then launch an emulator, boot its operating system, and start the main activity of

our application. Although, an emulator appearance may vary, the result will be similar to the one you see in Figure 15.8:

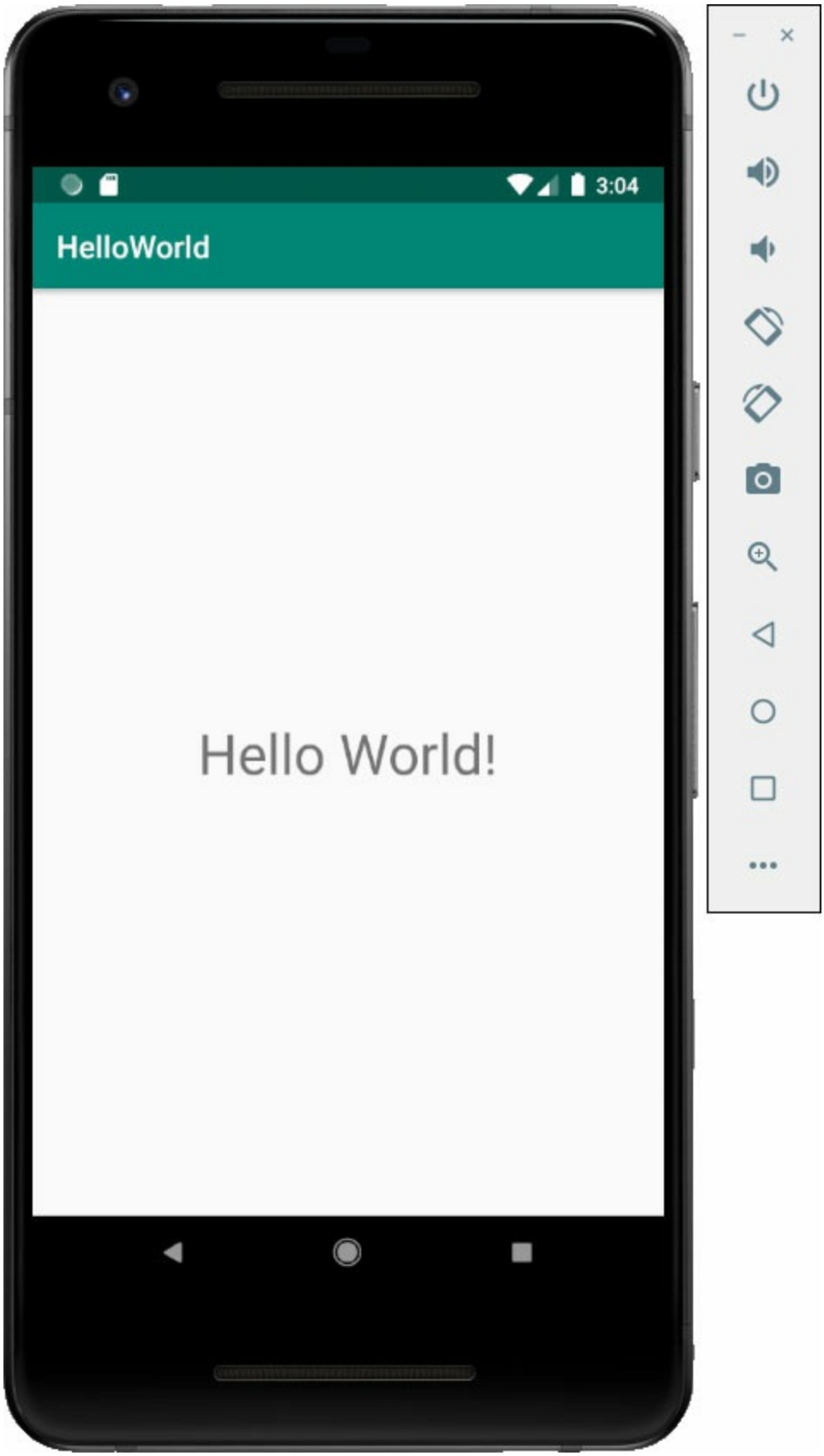
3:04
HelloWorld
Hello World!

Figure 15.8: *Running application on Android emulator*

You can interact with the emulator similarly to see how you do it with a physical smartphone or tablet. The side panel gives you access to some basic functions like device rotation, volume control, taking screenshots, and so on. It's also possible to debug applications deployed on an emulator. For this, you need to start the application in the **`debug`** mode using the Debug command instead of **`Run`**.

Now that we've got a basic understanding of our project structure, let's see how to make our application a bit more interactive.

Activities

The remaining part of the chapter will be centered on an example of a calculator application. In the course of this section, we'll see how to define the activity UI using both layout XML and Anko DSL, how synthetic properties of the Android Extensions plugin can help you simplify UI-related code as well as take a glimpse at the activity lifecycle, and get to know how you can preserve its state and you may need to do it.

Designing the application UI

We'll take our Hello, World example as a starting point for our calculator application. First, let's change the application title in the `strings.xml` resource file:

```
<resources>
<string name="app_name">Calculator</string>
</resources>
```

Now, let's open `activity_main.xml` which contains the UI definition of the main activity. Click on the **`Text`** tab below to bring up a text representation of the underlying file and edit it to match the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:id="@+id/relative1"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity">
```

```
<TableLayoutandroid:layout_width="match_parent"
android:layout_height="match_parent"
android:stretchColumns="3">
<TextViewandroid:id="@+id/txtResult"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:textSize="40sp"/>
<TableRowandroid:layout_width="match_parent"
android:layout_height="match_parent">
<Button android:id="@+id/btn7"
android:text="7"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btn8"
android:text="8"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btn9"
android:text="9"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btnPlus"
android:text="+"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="end|center_vertical"/>
</TableRow>
<TableRowandroid:layout_width="match_parent"
android:layout_height="match_parent">
<Button android:id="@+id/btn4"
android:text="4"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btn5"
android:text="5"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btn6"
android:text="6"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btnMinus"
android:text="-"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="end|center_vertical"/>
</TableRow>
<TableRowandroid:layout_width="match_parent"
```

```
android:layout_height="match_parent">
<Button android:id="@+id/btn1"
android:text="1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btn2"
android:text="2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btn3"
android:text="3"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btnTimes"
android:text="*"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="end|center_vertical"/>
</TableRow>
<TableRowandroid:layout_width="match_parent"
android:layout_height="match_parent">
<Button android:id="@+id/btn0"
android:text="0"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btnPoint"
android:text="."
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btnSign"
android:text="+/-"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btnDivide"
android:text="/"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="end|center_vertical"/>
</TableRow>
<TableRowandroid:layout_width="match_parent"
android:layout_height="match_parent">
<Button android:id="@+id/btnBackspace"
android:text="&lt;-"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btnClear"
android:text="C"
android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"/>
<Space android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
<Button android:id="@+id/btnCalc"
android:text="="
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="end|center_vertical"/>
</TableRow>
</TableLayout>
</RelativeLayout>
```

Figure 15.9 shows a preview of the calculator UI:

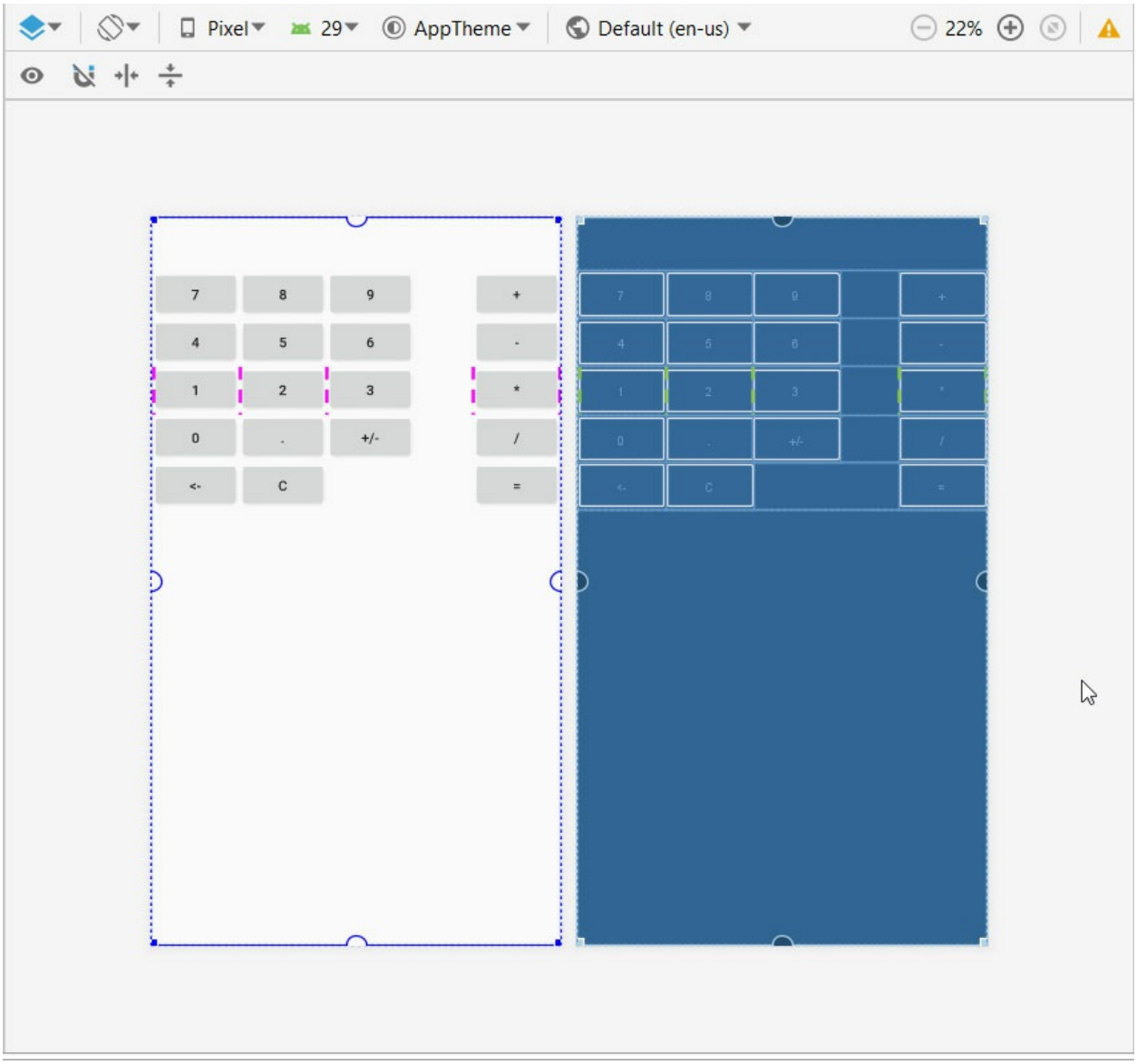

Figure 15.9: *Calculator UI preview*

Discussing the format of the layout XML is out of the scope of this book, so we won't delve into details here, but just point out some basics. The layouts are basically containers which arrange nested views in a certain way (you can compare them to Swing containers with a specific `LayoutManager`). Since the calculator UI assumes largely regular positioning of elements, we will use the table layout which assigns nested components to rows and columns. The components themselves are defined by such tags as **`Button`** (a simple button with a text) and **`TextView`** (a component which displays read-only text similar to Swing's **`JLabel`**).

Note the **`android:id`** attributes. They are assigned to each view element that presents or takes some data from the user. This attribute allows you to reference the corresponding element in Java or the Kotlin code. We'll see how to do it in the next section where we'll discuss the activity class.

Implementing the activity class

Let's get to the implementation of an activity class which adds some behavior to the UI. We won't go into details of the calculator business logic letting the readers to explore it by themselves but instead highlight some Android-specific points.

Among other things, we want to access the current value typed into the calculator's display: the `txtResult` component. To do this, we use the `findViewById()` function which passes a text view id. We don't need to put an actual string from the layout XML because such IDs can be referred to using the R class we've already seen in the Hello, World example:

```
private valtxtResult by lazy { findViewById<TextView>
(R.id.txtResult) }
```

Using the view reference, we can then access its members; for example, the `setOnClickListener()` method allows us to specify the action invoked when a user clicks on a button:

```
findViewById<Button>(R.id.btn0).setOnClickListener {
appendText("0") }
```

While reading/writing, the text property of a `TextView` component gives access to its text content:

```
private fun clearText() {
```

```
 txtResult.text = "0"
}
```

Here is the full source code of the calculator's `MainActivity` class:

```
package com.example.helloworld
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.TextView
import android.widget.Toast
import java.lang.ArithmeticException
import java.math.BigDecimal
import java.math.RoundingMode

class MainActivity :AppCompatActivity() {
 enum class OpKind {
  ADD, SUBTRACT, MULTIPLY, DIVIDE
  }

 companion object {
  fun OpKind.compute(a: BigDecimal, b: BigDecimal) = when (this)
  {
   OpKind.ADD -> a + b
   OpKind.SUBTRACT -> a - b
   OpKind.MULTIPLY -> a * b
   OpKind.DIVIDE ->a.divide(b, 10, RoundingMode.HALF_EVEN)
  }
 }

 private valtxtResult by lazy { findViewById<TextView>
 (R.id.txtResult) }

 private var lastResult: BigDecimal = BigDecimal.ZERO;
 private var lastOp: OpKind? = null
 private var waitingNextOperand: Boolean = false

 override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)
  findViewById<Button>(R.id.btn0)
  .setOnClickListener { appendText("0") }
  findViewById<Button>(R.id.btn1)
  .setOnClickListener { appendText("1") }
  findViewById<Button>(R.id.btn2)
  .setOnClickListener { appendText("2") }
  findViewById<Button>(R.id.btn3)
  .setOnClickListener { appendText("3") }
  findViewById<Button>(R.id.btn4)
```

```
  .setOnClickListener { appendText("4") }
  findViewById<Button>(R.id.btn5)
  .setOnClickListener { appendText("5") }

  findViewById<Button>(R.id.btn6)
  .setOnClickListener{ appendText("6") }
  findViewById<Button>(R.id.btn7)
  .setOnClickListener { appendText("7") }
  findViewById<Button>(R.id.btn8)
  .setOnClickListener { appendText("8") }
  findViewById<Button>(R.id.btn9)
  .setOnClickListener { appendText("9") }
  findViewById<Button>(R.id.btnPoint)
  .setOnClickListener { appendText(".") }
  findViewById<Button>(R.id.btnSign)
  .setOnClickListener {
   valcurrentText = txtResult.text.toString()
   txtResult.text = when {
    currentText.startsWith("-") ->
    currentText.substring(1, currentText.length)
    currentText != "0" -> "-$currentText"
    else ->return@setOnClickListener
    }
   }

  findViewById<Button>(R.id.btnBackspace)
  .setOnClickListener {
   valcurrentText = txtResult.text.toString()
   valnewText = currentText.substring(0, currentText.length - 1)
   txtResult.text =
   if (newText.isEmpty() || newText == "-") "0" else newText
  }

  findViewById<Button>(R.id.btnClear)
  .setOnClickListener { clearText() }

  findViewById<Button>(R.id.btnPlus)
  .setOnClickListener { calc(OpKind.ADD) }
  findViewById<Button>(R.id.btnMinus)
  .setOnClickListener { calc(OpKind.SUBTRACT) }
  findViewById<Button>(R.id.btnTimes)
  .setOnClickListener { calc(OpKind.MULTIPLY) }
  findViewById<Button>(R.id.btnDivide)
  .setOnClickListener { calc(OpKind.DIVIDE) }
  findViewById<Button>(R.id.btnCalc)
  .setOnClickListener { calc(null) }

  clearText()
}
```

```
 private fun clearText() {
  txtResult.text = "0"
 }

 private fun appendText(text: String) {
  if (waitingNextOperand) {
   clearText()
   waitingNextOperand = false
  }
   val currentText = txtResult.text.toString()
   txtResult.text =
   if (currentText == "0") text else currentText + text
 }

 private fun calc(nextOp: OpKind?) {
  if (waitingNextOperand) {
   lastOp = nextOp
   return
  }

  val currentValue = BigDecimal(txtResult.text.toString())
  val newValue = try {
   lastOp?.compute(lastResult, currentValue) ?: currentValue
  } catch (e: ArithmeticException) {
   lastOp = null
   waitingNextOperand = true
   Toast.makeText(
    applicationContext,
    "Invalid operation!",
    Toast.LENGTH_SHORT
    ).show()
   return
  }
  if (nextOp != null) {
   lastResult = newValue
  }
  if (lastOp != null) {
   txtResult.text = newValue.toPlainString()
  }
  lastOp = nextOp
  waitingNextOperand = nextOp != null
 }
}
```

Now, we can try to run the calculator and see it in action. Figure 15.10 shows an example of using our application in the Android emulator:

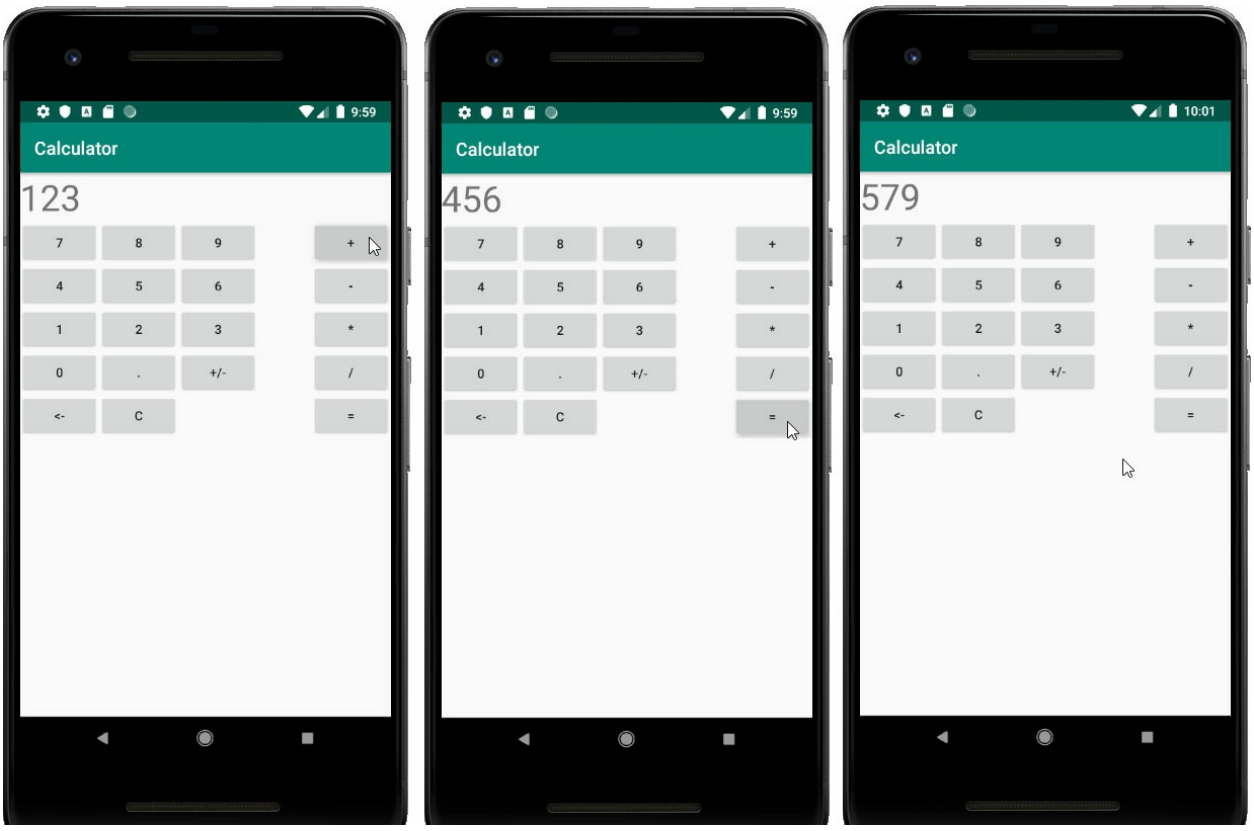

Figure 15.10: *Calculator application in action*

Kotlin Android Extensions

Frequent use of `findViewById()` can clutter your code, especially if you want to keep references to view components in class properties like we've done with `txtResult`. In the Java world, some libraries like `Butterknife` or Android Data Binding are able to work around this problem by automatically injecting view references into given class fields, but require you to manually annotate each field of interest specifying the corresponding ID. While these libraries can also be used in Kotlin, there is a much more concise solution which relies on an Android Extensions compiler plugin. In the preceding section, we've mentioned that this plugin is automatically activated in the Android Studio-generated project. Now, we'll finally put it to some real use.

The key feature of Kotlin Android Extensions is a set of synthetic properties corresponding to view components. Namely, they allow you to reference view using their identifiers as if they were names of some top-level properties. All you need to do is add an import directive of the form:

```
import kotlinx.android.synthetic.main.activity_main.*
```

Where `activity_main` is a file name of the activity XML file. Such a synthetic package contains a set of `Activity` extension properties, each corresponding to some component in the activity XML. For example, we can simplify the following line:

```
findViewById<Button>(R.id.btn0)
.setOnClickListener { appendText("0") }
to:
btn0.setOnClickListener { appendText("0") }
```

As a result, the `onCreate()` method of our calculator activity transforms into the following:

```
override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_main)

 btn0.setOnClickListener { appendText("0") }
 btn1.setOnClickListener { appendText("1") }
 btn2.setOnClickListener { appendText("2") }
 btn3.setOnClickListener { appendText("3") }
 btn4.setOnClickListener { appendText("4") }
 btn5.setOnClickListener { appendText("5") }
 btn6.setOnClickListener { appendText("6") }
 btn7.setOnClickListener { appendText("7") }
 btn8.setOnClickListener { appendText("8") }
 btn9.setOnClickListener { appendText("9") }
 btnPoint.setOnClickListener{ appendText(".") }
 btnSign.setOnClickListener{ … }

 btnBackspace.setOnClickListener{ … }
 btnClear.setOnClickListener{ clearText() }
 btnPlus.setOnClickListener{ calc(OpKind.ADD) }
 btnMinus.setOnClickListener{ calc(OpKind.SUBTRACT) }
 btnTimes.setOnClickListener{ calc(OpKind.MULTIPLY) }
 btnDivide.setOnClickListener{ calc(OpKind.DIVIDE) }
 btnCalc.setOnClickListener{ calc(null) }

 clearText()
}
The explicit txtResult property can be dropped altogether:
private valtxtResult by lazy { findViewById<TextView>
(R.id.txtResult) }
```

IDE Tips: Thanks to the auto-import feature available in both IntelliJ IDEA and Android Studio, you don’t need to add such imports by hand. If you try to reference the view component by its ID

and it hasn't imported the necessary synthetic property yet, the IDE will suggest to do it for you (see Figure 15.11 for example):

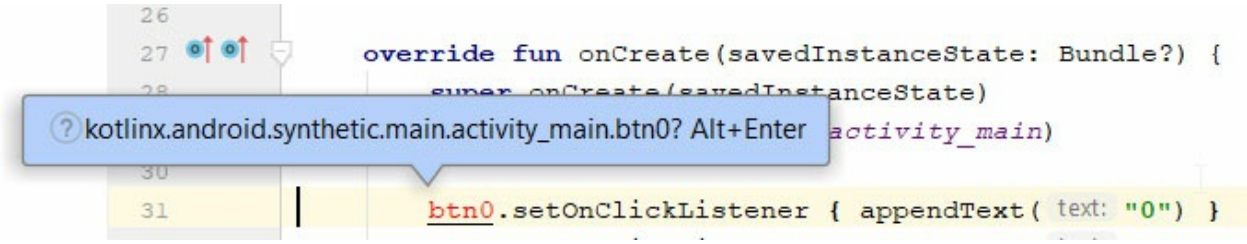

***Figure 15.11:** Automatic import of a synthetic property*

On top of that, Android Extensions provide you a bunch of useful features such as tuning a caching policy of views available via synthetic properties and the ability to enhance any container with a similar injection mechanism by implementing the `LayoutContainer` interface. We won't discuss these features in detail here, but you can find comprehensive information in the Android Extensions tutorial at **http://kotlinlang.org.**

Preserving the activity state

If you experiment a bit with the calculator we created in the previous sections, you may find out a visible flaw in its behavior. Let's type some number and then imitate the device rotation. To do this, you may just click on one of the `Rotate left/Rotate right` buttons on the emulator side panel. The result is shown in Figure 15.12:

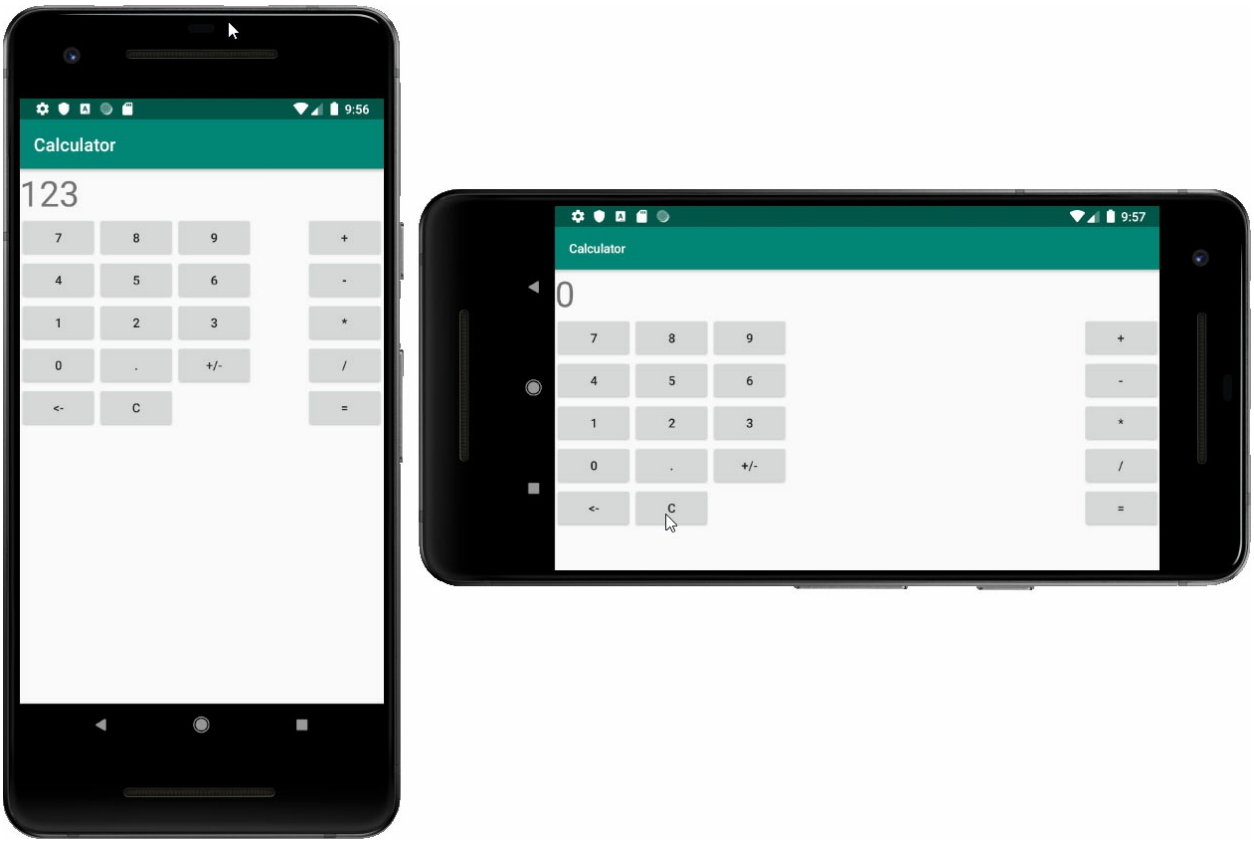

Figure 15.12: *Activity reset on device rotation*

You can see that an original number has changed to zero. In fact, the entire state of our activity has reset to the one we've provided in the initialization code. The reason of such a behavior is that change of device configuration (such as rotating its screen) forces the system to destroy our activity and recreate it from scratch. A similar case is when an activity is not visible to the user and the system runs resources necessary to execute the application with higher priority. As a result, Android may shut down the lesser-priority process together with its activities.

But what if need to retain some state between different instantiations of the same activity regardless of whether it was forcibly destroyed/recreated by the system? Android provides a solution in the form of a so-called, bundle which is basically a set of key/value pairs you can use to store any serializable data. As you must've noticed, the `onCreate()` method takes a Bundle parameter which contains data preserved from the previous activity run. To fill the bundle, we need to override another `Activity` method, `onSaveInstanceState()`.

For example, to fix the calculator behavior, we need to preserve the activity

state which in our case is composed of a text shown on the calculator display and values of instance variables, `lastResult`, `lastOp` and `waitingNextOperand`. The first step is to override the `onSaveInstanceState()` method where we write relevant values into a Bundle object:

```
override fun onSaveInstanceState(outState: Bundle) {
 super.onSaveInstanceState(outState)
 outState.putString("currentText", txtResult.text.toString())
 outState.putSerializable(::lastResult.name, lastResult)
 outState.putSerializable(::lastOp.name, lastOp)
 outState.putBoolean(::waitingNextOperand.name,
 waitingNextOperand)
}
```

Now, even if the activity is temporarily destroyed due to configuration change or the need to free device resources for other processes and recreated at some later time, the system preserves the bundle and passes it to the `onCreate()` method for initialization. The second step is to add a code which reads from a bundle into our `onCreate()` implementation:

```
override fun onCreate(savedInstanceState: Bundle?) {
 …
 clearText()

 savedInstanceState?.let {
  txtResult.text = it.getString("currentText")
  lastResult = it.getSerializable(::lastResult.name) as
  BigDecimal
  lastOp = it.getSerializable(::lastOp.name) as OpKind?
  waitingNextOperand = it.getBoolean(::waitingNextOperand.name)
 }
}
```

An argument passed to `onCreate()` may be null. This happens when the bundle is absent; for example, if the activity is started for the first time.

Note that only serializable values may be stored in a bundle. If you need to preserve some non-serializable objects, you need to either implement the `Serializable` interface, or refrain from writing it to the bundle directly and take an alternative approach such as converting it to some serializable data holder or preserving the original object by parts.

One more thing worth mentioning is that bundles are only suited for preserving relatively small amount of transient data because their

serialization occupies the main thread and consumes memory of the system process. For other cases, it's recommended to use local storage such as user preferences or SQLite database.

The `onCreate()` and `onSaveInstanceState()` methods are special cases of the so-called lifecycle callbacks which are invoked by the Android OS when an activity transitions to a new state from the lifecycle's view. For example, the `Resumed` state is associated with the activity running in the foreground, the `Paused` corresponds to the activity which is moved to the background but remains visible to the user, while the `Stopped` state means that the activity becomes completely invisible.

The overridden versions of lifecycle callbacks must invoke inherited implementations as well because they contain a common code necessary for proper functioning of the activity.

IDE Tips: Android Studio includes an inspection which reports an error if you override lifecycle callbacks without calling the inherited method (as shown in Figure 15.13):

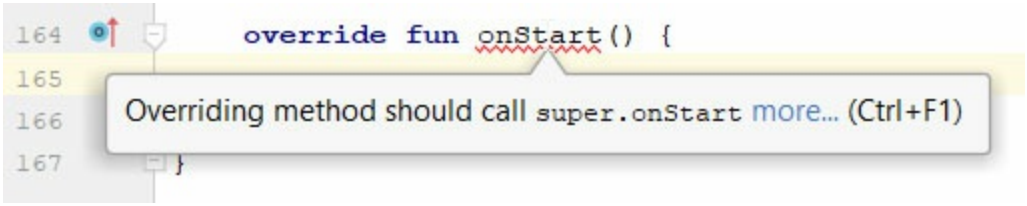

Figure 15.13: *Error on absent super call*

Anko Layouts

Anko is a Kotlin framework which provides various enhancements simplifying development of Android applications. Its major features include:

- A large set of helpers for working with dialogs, intents (objects which you can use to start new activities), logging and application resources – combined into Anko Commons library.
- A domain-specific language for description of the UI layout (Anko Layouts).
- API simplifying an access to SQLite databases which are commonly used for a local storage in Android (Anko SQLite).

In this section, we'll take a glimpse at the layout DSL you can use as a replacement of XML files with a description of view components.

Before we can make use of Anko features in our project, we have to configure the corresponding dependency in the module Gradle script. Let's add the following line to the dependencies block:

```
implementation "org.jetbrains.anko:anko:0.10.8"
```

When you make changes in one of the Gradle build files, the IDE will detect them and display a warning at the top of file editor (as shown in Figure 15.14):

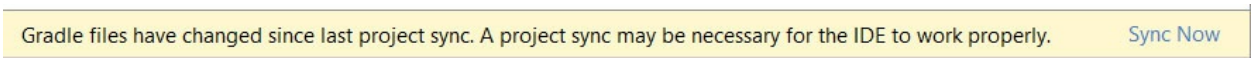

Figure 15.14: *Android Studio suggesting Gradle synchronization*

After you click on the **`Sync Now`** link, Android Studio will synchronize its internal project model with a new Gradle-provided configuration making Anko library available in various code insight features such as automatic completion and code navigation.

Now, we can rewrite our calculator layout using Anko DSL. The extra code will replace the `setCurrentView()` call at the start of the `onCreate()` method:

```
…
private fun _TableRow.charButton(text: String) {
 button(text).onClick { appendText(text) }
}

private fun _TableRow.opButton(op: OpKind?) {
 val text = when (op) {
  OpKind.ADD -> "+"
  OpKind.SUBTRACT -> "-"
  OpKind.MULTIPLY -> "*"
  OpKind.DIVIDE -> "/"
  null -> "="
 }
 button(text)
 .lparams { gravity = Gravity.END or Gravity.CENTER_VERTICAL }
 .onClick { calc(op) }
}
private lateinit var txtResult: TextView

override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
```

```
tableLayout {
 setColumnStretchable(3, true)
 txtResult = textView("0")
 .lparams(width = matchParent, height = wrapContent)
 .apply { textSize = 40.0f }
 tableRow {
  charButton("7")
  charButton("8")
  charButton("9")
  opButton(OpKind.ADD)
 }
 tableRow {
 charButton("4")
 charButton("5")
 charButton("6")
 opButton(OpKind.SUBTRACT)
 }
 tableRow {
 charButton("1")
 charButton("2")
 charButton("3")
 opButton(OpKind.MULTIPLY)
 }
 tableRow {
  charButton("0")
  charButton(".")
  button("+/-").onClick {
   val currentText = txtResult.text.toString()
   txtResult.text = when {
    currentText.startsWith("-") ->
    currentText.substring(1, currentText.length)
    currentText != "0" -> "-$currentText"
       else ->return@setOnClickListener
     }
  }
  opButton(OpKind.DIVIDE)
 }
 tableRow {
  button("<-").onClick {
   valcurrentText = txtResult.text.toString()
   valnewText = currentText.substring(0, currentText.length -
   1)
   txtResult.text =
    if (newText.isEmpty() || newText == "-") "0" else newText
  }
  button("C").onClick { clearText() }
  space()
  opButton(null)
```

```
    }
  }
}
…
```

Make sure to also add an import directive for Anko-specific declarations:

```
import org.jetbrains.anko.*
```

It's not hard to see that the UI code structure basically parallels that of the layout XML. We will also introduce two helper functions, `charButton()` and `opButton()` which define character and operation buttons, respectively. We can also extract the layout code to separate the file and use it like an external UI definition script. Since it's still a Kotlin code, we can easily reuse its code and subject it to various refactorings. The original `activity_main.xml` can be safely dropped.

More details about the framework features as well as comprehensive documentation can be found on the AnkoGitHub site: **https://github.com/Kotlin/anko**.

Conclusion

In this chapter, you learned how to use basic features of the Android Studio IDE and created your first Android application using the Kotlin language. We introduced you to a concept of Android activity, got a taste of the UI layout description and demonstrated how Android Extensions and Anko can help you in writing UI-related Kotlin code.

In the next chapter, we will learn how to develop web applications using the Ktor framework. We'll talk about basic Ktor features, project setup and how to use routing mechanism for handling client requests. We'll also take a look at DSL aimed at generation of HTML content.

Questions

1. Describe the project setup in Android Studio.
2. Describe the use of Gradle for project configuration in Android Studio. How do you add a new dependency?
3. How do you configure a virtual device for running applications?

4. What is an activity? How do you describe an UI of an Android application?
5. Explain the synthetic properties introduced by Kotlin Android Extensions.
6. How do you save/restore an activity state when it gets temporarily destroyed?
7. What is Anko? Describe the UI layout DSL available in the Anko framework.

CHAPTER 16

Web Development with Ktor

Web development comprises a major part of modern software engineering with applications ranging from full-fledged enterprise systems with rich UI and complex workflow to task-specific microservices. Most of the frameworks aimed at simplifying various aspects of web developments in the Java world can be easily used in the Kotlin environment thanks to Java/Kotlin interoperability. A special mention, however, should be made of the tools designed with out-of-the-box Kotlin support because they allow Kotlin developers to derive maximum benefit of the language feature, increasing one's productivity.

In this chapter, we'll introduce you to one of such Kotlin-specific frameworks which is called Ktor. This is a Kotlin framework with a goal to simplify development of connected systems which may include various client and server applications such as browsers, mobile clients, web applications, and services. Being an extension of the Coroutines library, Ktor offers powerful and easy-to use facilities for asynchronous communication. This material is certainly not meant to be an exhaustive treatment of Ktor capabilities, so in the scope of our book, we'll limit this discussion to a small set of features related to web applications, especially their server-side part; dispatching client requests, and obtaining request data, and composing various kinds of responses. The readers are encouraged to continue their acquaintance with Ktor at its official website **https://ktor.io and other resources.**

Structure

- Setting up a Ktor project
- Server features
- Client features

Objective

Learn basic features of using Ktor for client-and server-side development of web applications.

Introducing Ktor

Ktor is a framework for development of client and server applications which communicate with each other over the network. Although, one of its main applications revolves around web applications using the HTTP protocol for data exchange, the ultimate Ktor goal is to become a general-purpose multi-platform framework for building all kinds of connected applications. Its designer and primary developer is JetBrains which is also responsible for the creation of the Kotlin language itself. The key features of Ktor as compared to various J2EE frameworks are as follows:

- The use of Kotlin-powered domain-specific languages which allow you to use concise, declarative-style descriptions for some application aspects (such as request routing rules or HTML content) by easily combining them with the rest of your code.
- Out-of-the-box support of efficient asynchronous computations provided by the Kotlin coroutines library.

In this section, we'll take a quick glance at Ktor and walk you through the basic steps required for setting up a project in IntelliJ IDEA. The process can be simplified by making use of a special IntelliJ plugin which adds Ktor support to the project wizard. To install the plugin, follow the given steps:

1. Choose the **`File | Settings…`** command in the IDEA menu and click on the **`Plugins`** section on the left. You'll be presented with a **`Plugin Manager`** view.
2. Make sure that the **`Marketplace`** option is chosen on the top and type `ktor` in the top text field. You should see the Ktor plugin in the search results (as shown in Figure 16.1).
3. Now, click on the **`Install`** button and the IDE will automatically download the plugin and proceed with its installation. When the installation is complete, you'll be prompted to restart the IDE to activate changes in plugins. Confirm and wait until the IDE restart completes.

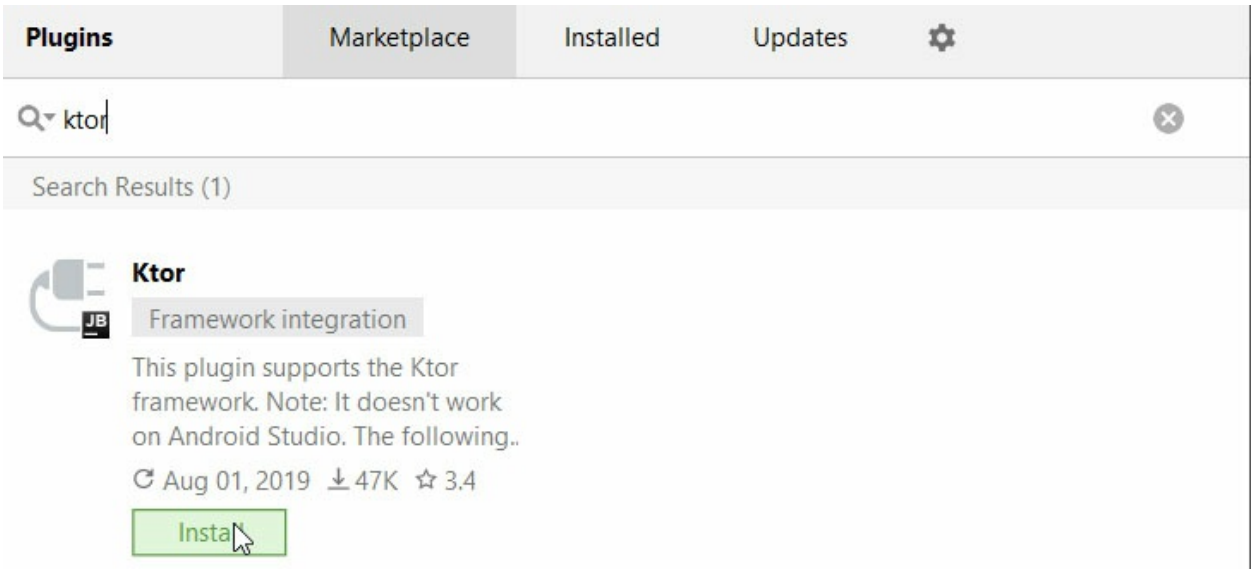

Figure 16.1: *Installing the Kotlin plugin for IntelliJ IDEA*

IDE Tips: Unlike most IntelliJ plugins for the Web/J2EE development, Ktor doesn't require the Ultimate edition and can be used in the IDEA Community as well.

Now, we can create a project. Choose the **`File | New | Project…`** command and select the **`Ktor`** option in the **`New Project`** dialog box. Apart from the basic client/server functionality, Ktor provides a set of pluggable features which require dependencies on additional artifacts and/or some configuration code. The first wizard step, as shown in *Figure 16.2*, allows you to choose the necessary features for both client-and server-side parts of your application. The IDE will automatically add all the necessary dependencies and generate a sample code demonstrating the feature usage:

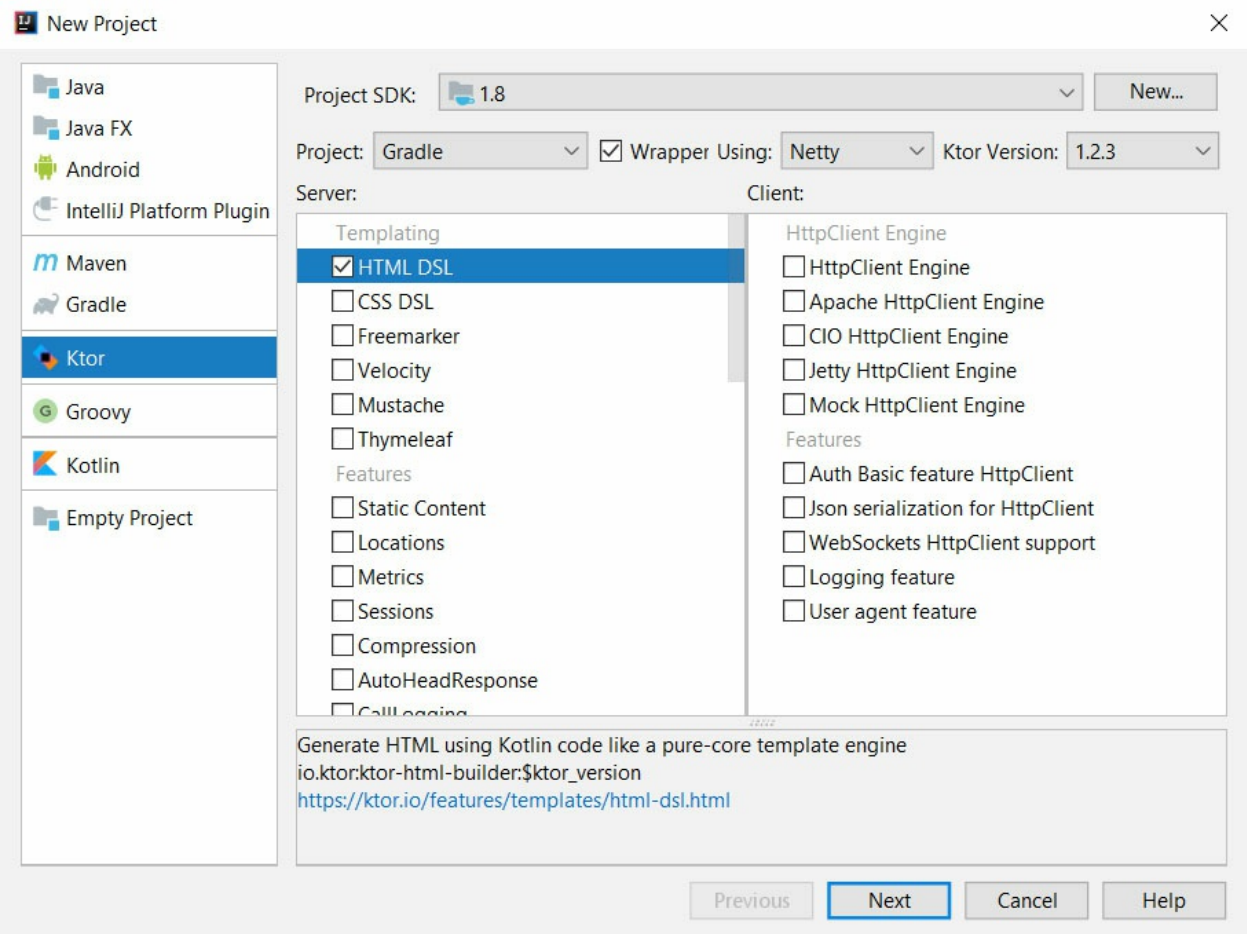

Figure 16.2: *Ktor project wizard*

For our example, we'll choose the HTML DSL option in the **`Server`** column which gives us the ability to generate the HTML markup using simple Kotlin DSL. Since we do not need any HTTP client functionality here, (basically we're not going to send requests to other servers), we leave the **`Client`** column unfilled.

Apart from specific features, Ktor also allows you to choose a project build system (such as Gradle or Maven), type of HTTP server engine (Netty/Jetty/Tomcat/Coroutine-based), and a framework version. Our project will be Gradle-based and use Ktor 1.2.3 with Netty engine. After making sure that all options are set correctly, click on **`Next`**.

In the next step, the IDE will ask you to fill basic information necessary for building project artifacts (group/artifact ID and version). You can leave these values as is and click on **`Next`**.

One more wizard step will require you to choose a project name and location.

After you click on **`Finish`**, the IDE will proceed with generation of project sources and open a new project on completion.

In the last step, you'll be presented with the **`Import Module from Gradle`** dialog where you can set up basic options for Gradle/IntelliJ interoperability. For now, let's choose the **`Use default gradle wrapper`** option and leave all other settings as is. After you click on **`OK`**, the IDE will start the process of synchronization with the Gradle project model. Once it completes, you'll be able to see the following files in the IDE **`Project View`** (Figure 16.3):

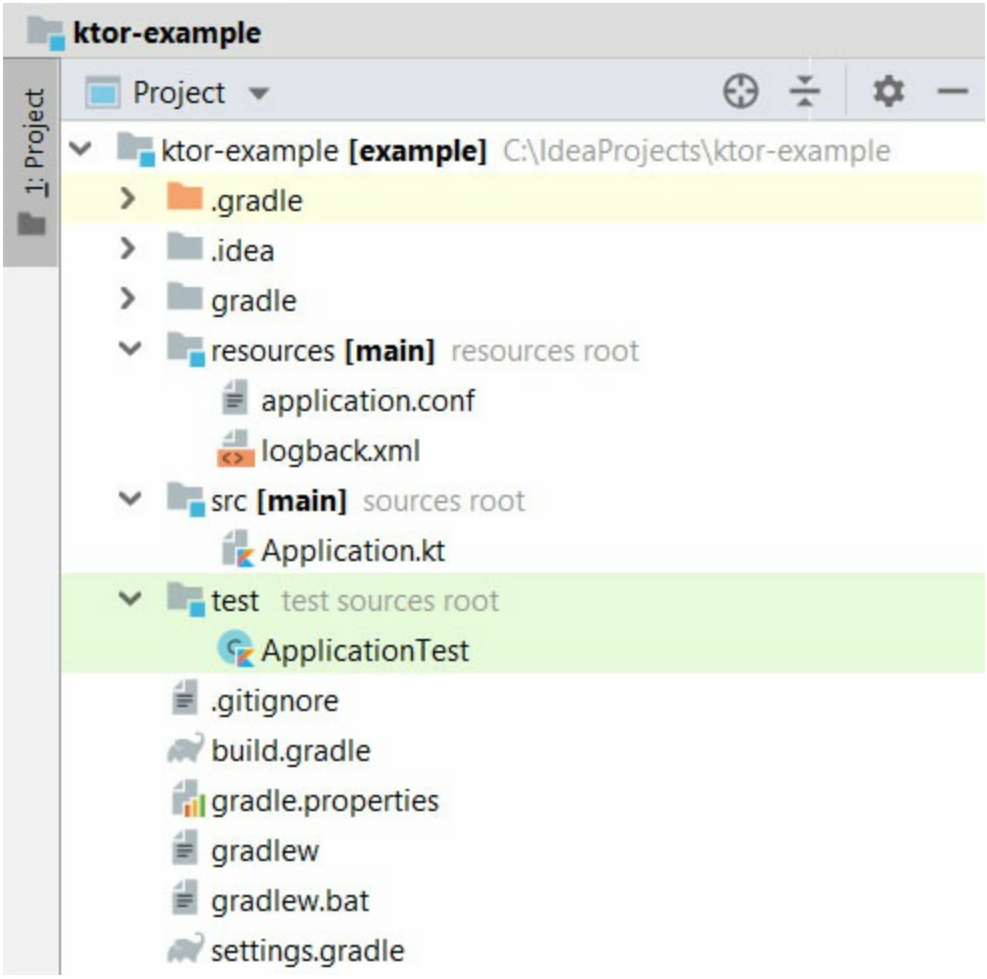

Figure 16.3: *Structure of a sample Ktor project*

Note the `application.conf` file in the resources folder. This file contains the configuration of your server application in a **HOCON (Human-Optimized Config Object Notation)** format. In our case, it specifies the server port number and a list of modules composing our application:

```
ktor {
```

```
 deployment {
  port = 8080
  port = ${?PORT}
 }
 application {
  modules = [ com.example.ApplicationKt.module ]
 }
}
```

Detailed information about HOCON and supported configuration options can be found on the Ktor site at **https://ktor.io/servers/configuration.html**.

The Ktor module is basically an extension function of the `Application` class which is responsible for configuring features, handling client requests, and other server tasks. Modules to be loaded must be specified by their qualified names in the server configuration file. Our sample project contains a single module implemented in `Application.kt`:

```
package com.example

import io.ktor.application.*
import io.ktor.client.HttpClient
import io.ktor.client.engine.apache.Apache
import io.ktor.html.respondHtml
import io.ktor.http.ContentType
import io.ktor.response.respondText
import io.ktor.routing.*
import kotlinx.html.*

fun main(args: Array<String>): Unit =
io.ktor.server.netty.EngineMain.main(args)

@Suppress("unused") // Referenced in application.conf
@kotlin.jvm.JvmOverloads
fun Application.module(testing: Boolean = false) {
 routing {
  get("/") {
   call.respondText(
    "HELLO WORLD!",
    contentType = ContentType.Text.Plain
   )
  }
  get("/html-dsl") {
   call.respondHtml {
    body {
     h1 { +"HTML" }
     ul {
      for (n in 1..10) {
```

```
            li { +"$n" }
          }
        }
      }
    }
  }
 }
}
```

The `main()` function which is also defined in this file simply starts the chosen HTTP server engine (in our case, it's Netty) which then reads `application.conf` and loads the server module represented by the `Application.module()` function.

The module body contains the routing block which sets up rules for handling client requests based on their URL path. In particular, when a client application (e.g. web browser) accesses the root path of our server, Ktor invokes the following handler:

```
call.respondText(
 "HELLO WORLD!",
 contentType = ContentType.Text.Plain
)
```

This code generates the HTTP response with a plain text message for the body. If you compile and start the server application and then open `localhost:8080` in some browser, you'll see something similar to *Figure 16.4*:

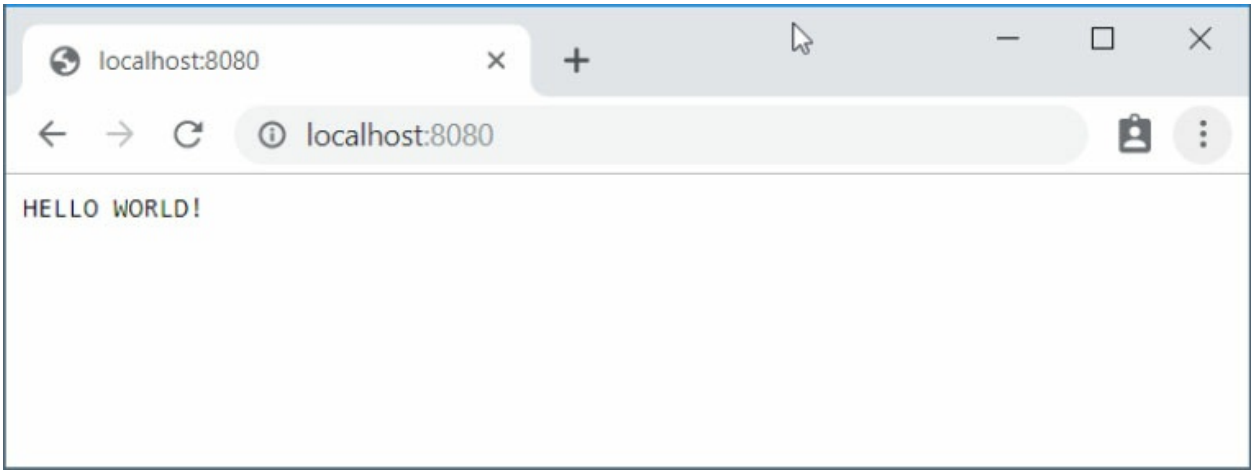

Figure 16.4: *Accessing the root path of Hello, World application in a browser*

Similarly, the `/html-dsl` path shows an example of generating a server

response using the HTML DSL library. This DSL allows you to present the HTML markup in the form of nested blocks corresponding to different HTML tags. As you've probably guessed the following handler:

```
call.respondHtml {
  body {
    h1 { +"HTML" }
    ul {
      for (n in 1..10) {
        li { +"$n" }
      }
    }
  }
}
```

Generates the HTML page with level 1 heading and a bullet list of numbers. Figure 16.5 shows the result of rendering the DSL code into an HTML page:

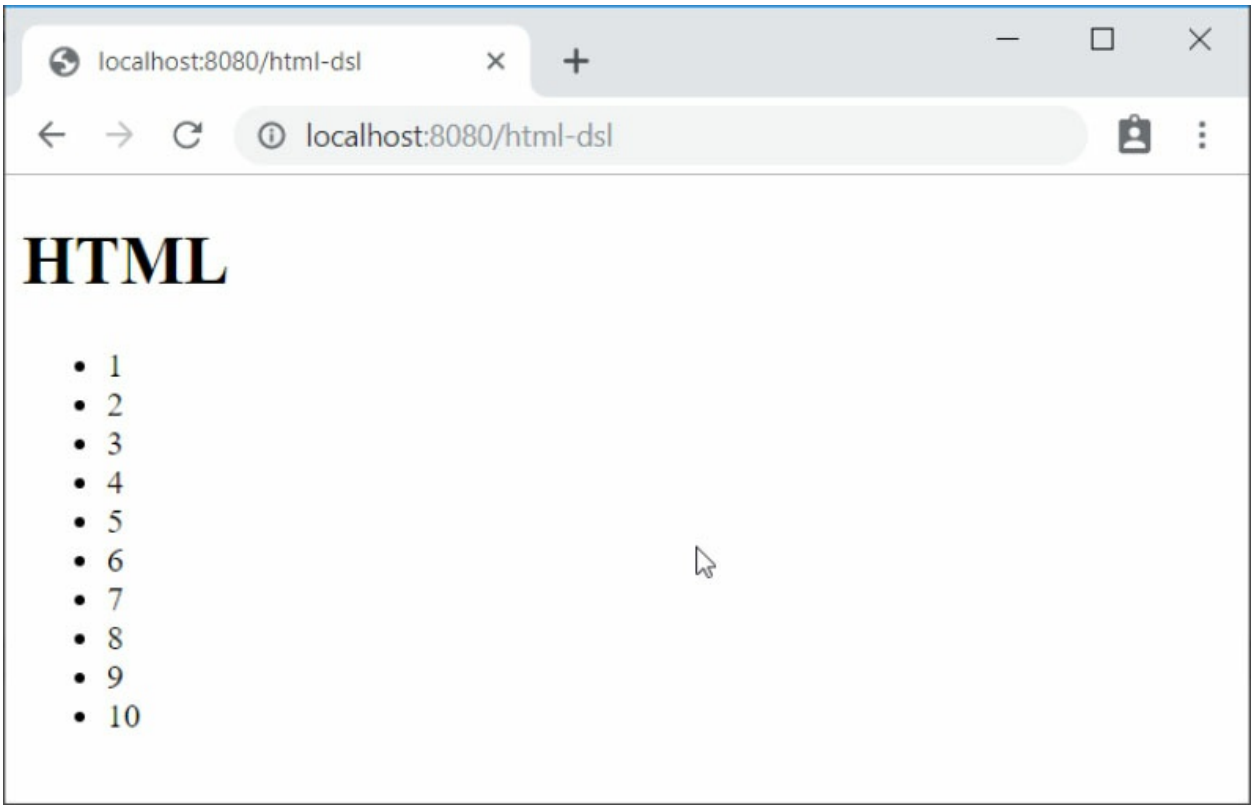

Figure 16.5: *HTML response associate with /html-dsl path*

In the upcoming section, we'll take a closer look at both HTML DSL and routing rules as well as some other server features of Ktor.

The IDE-generated project also includes a sample server test. Let's open the

`ApplicationTest` class:

```
package com.example

import io.ktor.http.*
import io.ktor.server.testing.*
import kotlin.test.*

class ApplicationTest {
 @Test
 fun testRoot() {
  withTestApplication({ module(testing = true) }) {
   handleRequest(HttpMethod.Get, “/”).apply {
   assertEquals(HttpStatusCode.OK, response.status())
    assertEquals(“HELLO WORLD!”, response.content)
   }
  }
 }
}
```

This code configures the test application for running with a given set of modules (note the use of the `testing = true` argument which allows the module code to distinguish test and production environments) and then checks the results of handling the simple HTTP request to the root path. You can use this class as a starting point for writing your own tests covering various aspects of the server behavior.

As an alternative to an IntelliJ plugin, you can make use of the online project generator available at **https://start.ktor.io**. The generator UI allows you to specify the same basic options, including a set of client/server features you’d like to use in your application. After you click on the **`Build`** button, the backend will suggest you to download an archive file containing a generated project. Figure 16.6 shows an example:

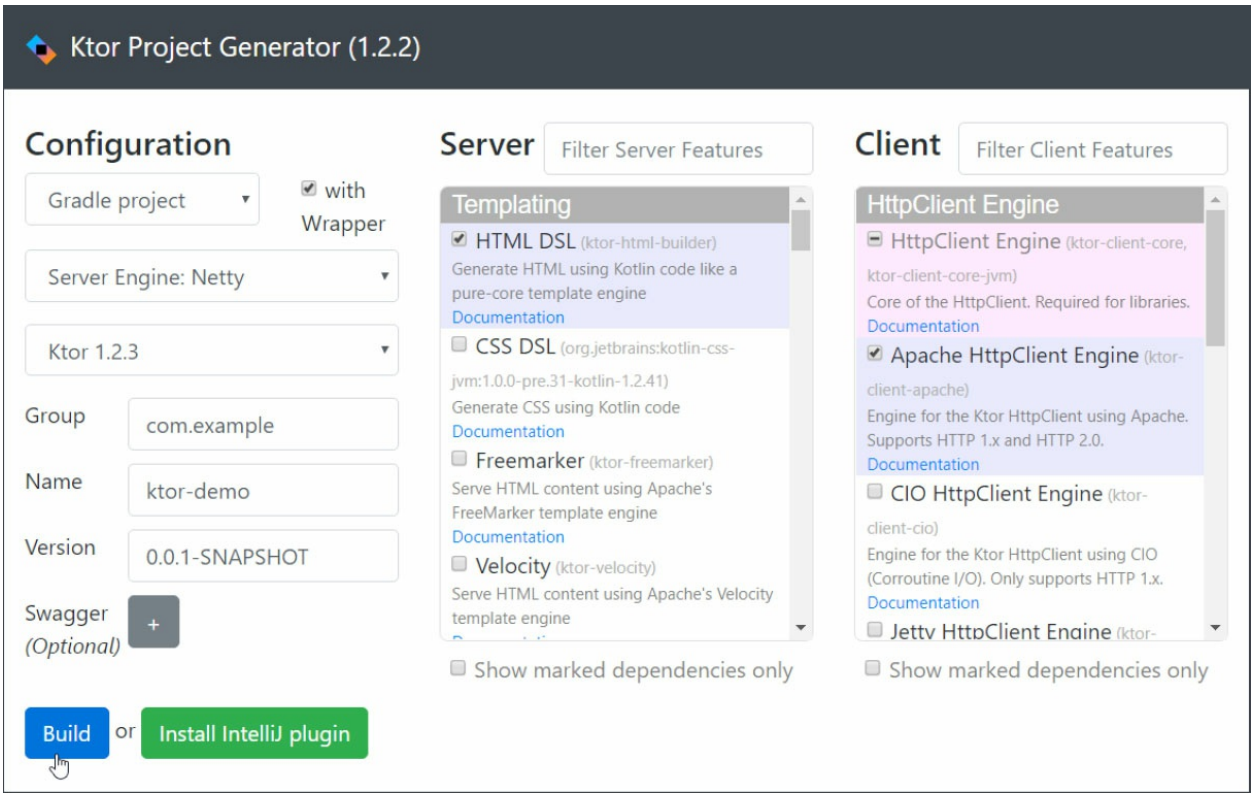

Figure 16.6: *Ktor project wizard at start.ktor.io*

This concludes our introduction to the basic project setup. In the upcoming sections, we'll focus on specific features provided by both client-and server-sides of the Ktor framework.

Server features

In this section, we'll consider a few topics regarding development of web server applications. Most of the Ktor functionality is organized as a set of pluggable features which can be configured by the call of the `install()` method which takes a feature object and an optional configuration block. For example, to enable compression of outgoing content, you can add the following code to the module function:

```
install(Compression)
```

If you want to specify additional (feature-specific) options such as choose a compression method, you may do so in the configuration block:

```
install(Compression) {
  gzip()
}
```

The examples in this chapter use an embedded HTTP server which allows them to run as a standalone program. In many cases, you may need to deploy the Ktor application in the context of some web/application container such as Apache Tomcat, Docker, or Google App Engine. To install your application, you'll need to assemble an archive which contains application classes together with all its dependencies and prepare container-specific configuration files. Discussion of container-specific details is beyond the scope of this book, but you can find detailed instructions on the official Ktor site at **https://ktor.io/servers/deploy.html.**

Routing

The routing feature allows you to implement a structured handling of HTTP requests based on the hierarchical system of pattern matchers. The routing configuration is expressed by a special **DSL (domain-specific language)** inside the feature installation block:

```
fun Application.module() {
 install(Routing) {
  // routine description
  get("/") { call.respondText("This is root page") }
 }
}
```

Or a `routing()` block which serves as a shorthand for the corresponding `install()` call:

```
fun Application.module() {
 routing {
  get("/") { call.respondText("This is root page") }
 }
}
```

The simplest routing scenario is given by the `get()` function which tells the server to execute a given handler for any HTTP `GET` request with a given URL path prefix. For example, the preceding code responds with a plain text string to the `GET` request at the site root while requests to any other path on the same site or with any other HTTP verb will result in 404 (see an example in Figure 16.7):

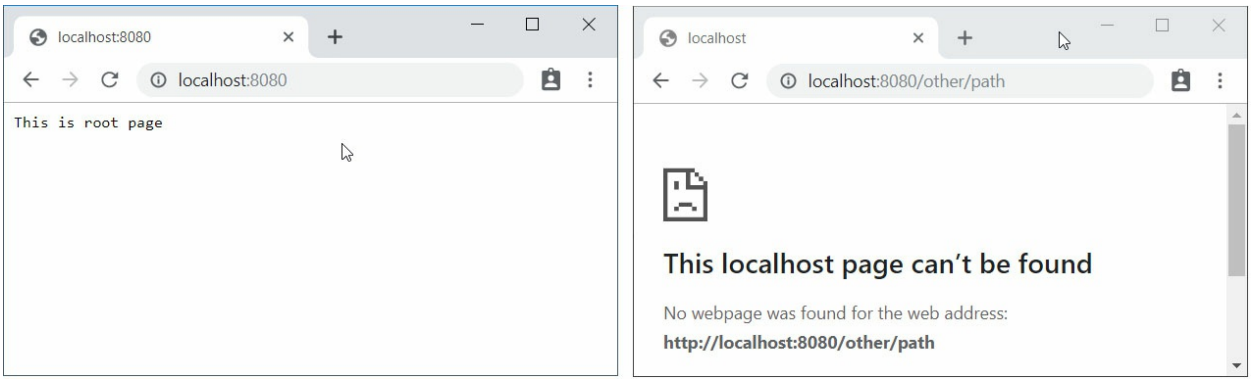

Figure 16.7: *Results of get("/") routing*

Ktor supports similar routing functions for all HTTP verbs, including `post()`, `put()`, `delete()`, `patch()`, `head()`, and `options()`.

Paths mentioned in routing functions may use parameters which match a specific segment of the request path and can be retrieved later from the application call. To introduce a parameter, you just need to enclose its name in braces. For example, the routing below will match any URL which starts with /`hello`/ and contains exactly two segments:

```
routing {
 get("/hello/{userName}") {
  call.respondHtml {
   body {
    h1 { +"Hello, ${call.parameters["userName"]}" }
   }
  }
 }
}
```

A parameter by itself can match just one segment, so paths like /`hello` or /`hello/John/Doe` will remain unmatched (as shown in Figure 16.8)

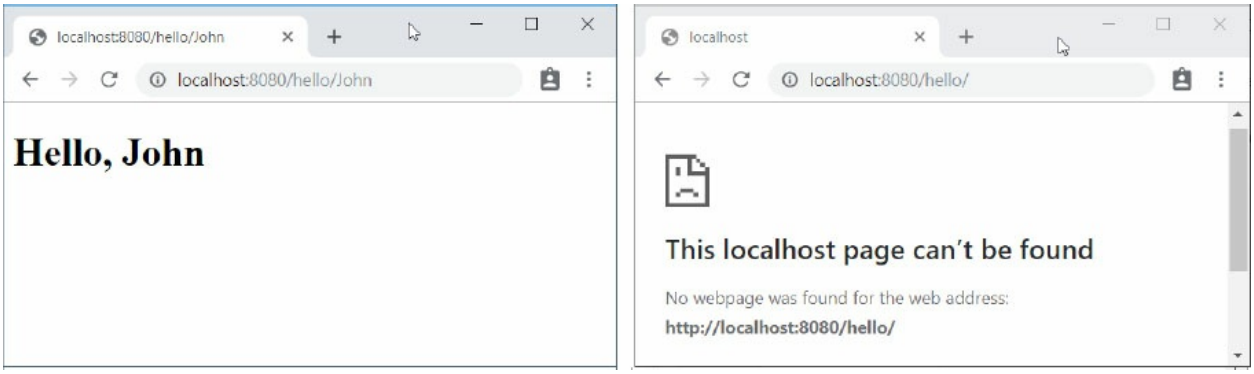

Figure 16.8: *Matching by a single-segment path parameter*

If the parameter value is not really used, you can replace it by a * character (wildcard):

```
routing {
 get(“/hello/*”) {
  call.respondHtml {
   body {
    h1 { +”Hello, World” }
   }
  }
 }
}
```

The routing above accepts the same set of paths as the previous one without capturing any parameters.

If you want to introduce an optional parameter which may match an empty path segment, just add ? to its name. The following routing will accept both `/hello/John` and `/hello` URLs as shown in Figure 16.9::

```
routing {
 get(“/hello/{userName?}”) {
  val userName = call.parameters[“userName”] ?: “Unknown”
  call.respondHtml {
   body {
    h1 { +”Hello, $userName” }
   }
  }
 }
}
```

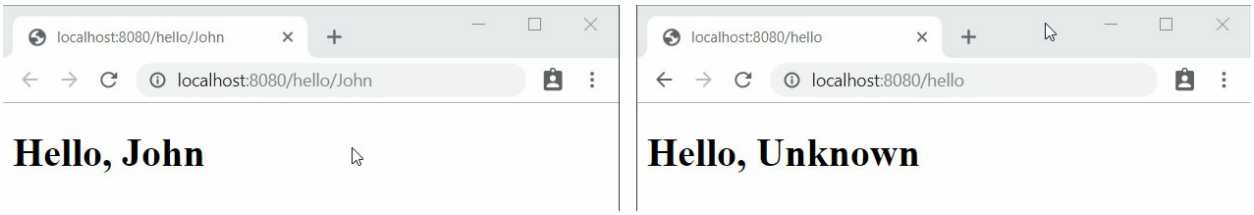

Figure 16.9: *Optional matching*

The tailcard … placed after the parameter name will match all URL segments at the path end. In this case, you can make use of `getAll()` methods of the `Parameters` class to obtain all captured segments split into a `List`. Let’s, for example, create a simple service which performs basic arithmetic operations on integer numbers and accepts input data in the form of URL paths such as `/calc/+/123/456`:

```
routing {
 get("/calc/{data…}") {
  val data = call.parameters.getAll("data") ?: emptyList()
  call.respondHtml {
   body {
    h1 {
     if (data.size != 3) {
      +"Invalid data"
      return@h1
     }
     val (op, argStr1, argStr2) = data
     val arg1 = argStr1.toBigIntegerOrNull()
     val arg2 = argStr2.toBigIntegerOrNull()
     if (arg1 == null || arg2 == null) {
      +"Integer numbers expected"
      return@h1
     }
     val result = when (op) {
      "+" -> arg1 + arg2
      "-" -> arg1 - arg2
      "*" -> arg1 * arg2
      "/" -> arg1 / arg2
      else -> null
     }
     +(result?.toString() ?: "Invalid operation")
    }
   }
  }
 }
}
```

You can see some of its responses in Figure 16.10. The top-left image shows an example of a successful response given for the /`calc/+/12345/67890` path which conforms to the expected format with an operation sign and pair of integers. The top-right image corresponds to the `Invalid data` case due to passing three numbers instead of two while the bottom-left image reports an error for an unknown operation **. Finally, the bottom-right image depicts a case of passing non-integer values for operands:

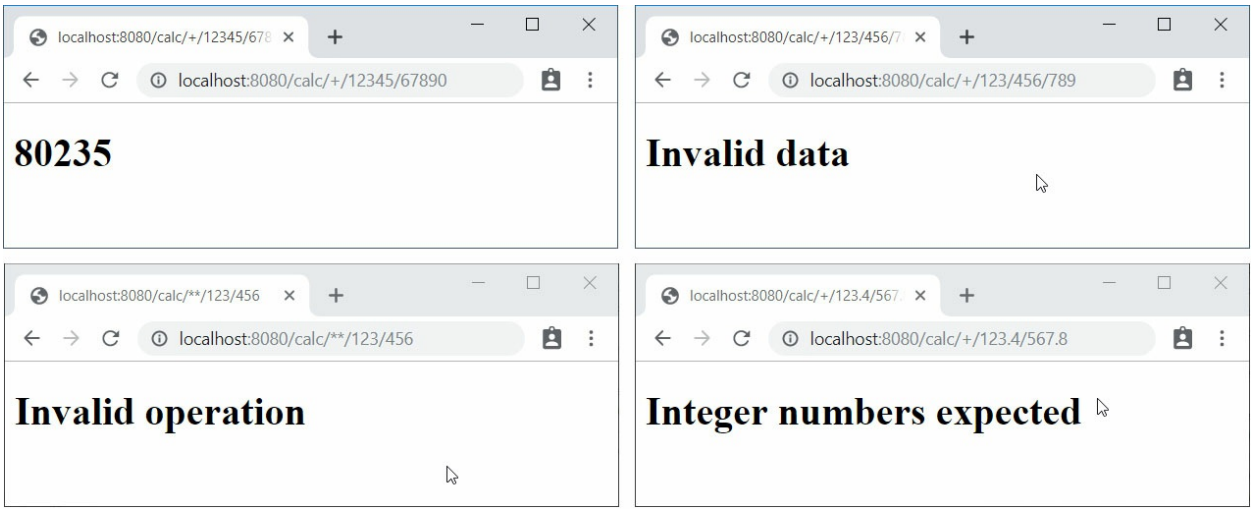

Figure 16.10: *Tailcard matching*

It's worth pointing out that tailcard matches empty path tails as well, so `get("/calc/{data…}")` the preceding handler is also invoked for the /calc path. The special `{…}` tailcard accepts all the remaining path tails but doesn't capture any parameters just like the * wildcard. For example, the handler like `get("/calc/{…}")` will match all paths starting with `/calc`, including `/calc` itself.

Apart from `get()`-like functions which match the entire URL path, Ktor allows you to define a routing tree matching subsequent portions of the URL and/or various request data. Consider the following example:

```
routing {
 method(HttpMethod.Get) {
  route("user/{name}") {
   route("sayHello") {
    handle {
     call.respondText("Hello, ${call.parameters["name"]}")
    }
   }
   route("sayBye") {
    handle {
     call.respondText("Bye, ${call.parameters["name"]}")
    }
   }
  }
 }
}
```

When the server receives an HTTP request it starts looking for a matching

routing rule starting from the tree root. In our case, the root node is `method(HttpMethod.Get)` which matches any request with the `GET` verb. If the client request satisfies this condition, the server goes down the tree and checks the `route("user/{name}")` rule which accepts URLs with a given path prefix. If the path fits, the server goes deeper and checks one of `route("sayHello")` and `route("sayBye")` rules which check the remaining portion of the URL path. At the lowest level, we have `handle()` blocks which generate responses provided that all rules on the corresponding branch are matched.

The HTTP verb can be specified in the `route()` call without explicit use of the `method()` function. The preceding `get()` function is basically a shorthand for the top-level `route()` block with a handler. For example, the following code:

```
routing {
 get("/hello/{userName}") {
  call.respondText("Hello, ${call.parameters["userName"]}")
 }
}
is equivalent to:
routing {
 route("/hello/{userName}", HttpMethod.Get) {
  handle {
   call.respondText("Hello, ${call.parameters["userName"]}")
  }
 }
}
```

The `route()` and `method()` are not the only matchers at your disposal. There are additional builders as well:

- `header(name, value)`: Accepts requests with a specific header
- `param(name, value)`: Accepts requests with a specific parameter value
- `param(name)`: Accepts requests which have a parameter with a specific name and captures its value
- `optionalParam(name)`: Accepts optional parameters with a given name

In the following example, we choose a response based on the value of action parameter:

```
routing {
 route("/user/{name}", HttpMethod.Get) {
```

```
    param("action", "sayHello") {
      handle {
        call.respondHtml {
          body { h2 { +"Hello, ${call.parameters["name"]}" } }
        }
      }
    }
    param("action", "sayBye") {
      handle {
        call.respondHtml {
          body { h2 { +"Bye, ${call.parameters["name"]}" } }
        }
      }
    }
  }
}
```

For example, the path `/user/John?action=sayHello` will produce **`Hello, John`** as the server response while `/user/John?action=sayBye` will give **`Bye, John`**. Any other action will remain unmatched, thus producing a **`not found`** response. Figure 16.11 demonstrates the results:

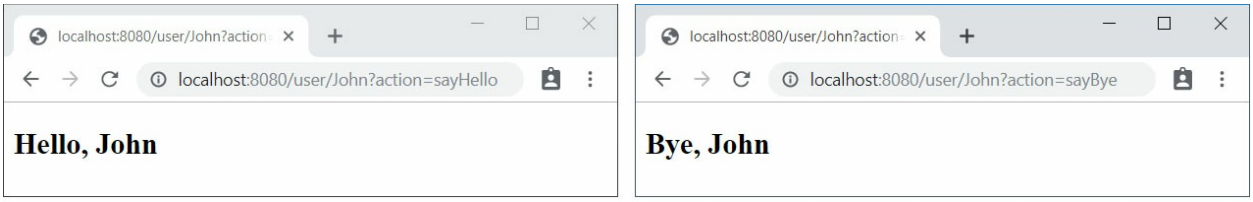

Figure 16.11: *Matching by request parameter*

The Ktor API allows you to create your own matchers, thus extending routing DSL. To do this, you need to provide an implementation of the `RouteSelector` class and add the corresponding builder function.

Handling HTTP requests

The main job of the HTTP server is to provide appropriate responses for client requests. To simplify this task, Ktor gives you a flexible and easy-to-use API which allows you to easily access request data such as URL path components, parameters, headers (both common and custom), and various content types of the request body as well generate response data. All you're left to do is to implement an actual processing logic with all low-level details hidden behind the framework.

In this and the following sections, we'll demonstrate basic capabilities of request/response processing in Ktor. We've already seen various examples of simple response generation via `responseText()` or `respondHtml()` inside a routing handler. Consider, for example, the following code:

```
routing {
 get("/") { call.respondText("This is root page") }
}
```

The call property available inside this handler is an instance of `ApplicationCall` which basically combines an incoming request with a response to be composed. A common case is building a response based on a text which is handled by the `responseText()` function. In the preceding example, we're sending a simple plain text body, but you can also specify the body MIME type using the `contentType` parameter as shown in the following code:

```
call.respondText("<h2>HTML Text</h2>", ContentType.Text.Html
```

This sends an HTML-based response. As an alternative, the response text can be provided by a suspending lambda:

```
call.respondText(ContentType.Text.CSS) { "p { color: red; }" }
```

You can also compose the body text using `PrintWriter`:

```
call.respondTextWriter(ContentType.Text.Html) {
 write("<head><title>Sample page</title><title>")
 write("<body><h2>Sample page</h2></body>")
}
```

To send an arbitrary binary data, you can use the `respondBytes()` function which takes `ByteArray` instead of a `String`:

```
get("/") {
 val data = "<h2>HTML Text</h2>".toByteArray()
 call.respondBytes(data, ContentType.Text.Html)
}
```

The `respondFile()` function can be used to transfer a file from the server to the client:

```
get("/{fileName}") {
 val rootDir = File("contentDir")
 val fileName = call.parameters["fileName"]!!
 call.respondFile(rootDir, fileName)
```

```
}
```

In addition to the body, you can set a response header data using the `call.response` property:

- `status(code: HttpStatusCode)`: Sets the HTTP response status
- `header(name: String, value: String)`: Appends the given header to the HTTP response

Ktor supports automatic redirection responses with status `301` (moved permanently) or `302` (moved temporarily):

```
routing {
 get(“/”) {
  call.respondRedirect(“index”)
 }
 get(“index”) {
  call.respondText(“Main page”)
 }
}
```

To access a request parameter, you can use the `request.queryParameters` object which serves as a kind of map from parameter names to their values. Suppose that we want to return a sum of a pair of integers given a URL of the form `/sum?left=2&right=3`. In this, we can obtain the values of left and right parameters by simply using the indexing operator on the `queryParameters` object:

```
routing {
 // e.g. /sum?left=2&right=3 responds with 5
 get(“/sum”) {
  val left = call.request.queryParameters[“left”]?.toIntOrNull()
  val right =
  call.request.queryParameters[“right”]?.toIntOrNull()
  if (left != null && right != null) {
   call.respondText(“${left + right}”)
  } else {
   call.respondText(“Invalid arguments”)
  }
 }
}
```

When a parameter is used more than one time, the `get()` function only returns its first value. The `getAll()` function, on the contrary, returns you a full list of parameter values in the form of `List<String>`:

```
routing {
 // e.g. /sum?arg=1&arg=2&arg=3 responds with 6
 get("/sum") {
  val args = call.request.queryParameters.getAll("arg")
  if (args == null) {
   call.respondText("No data")
   return@get
  }
  var sum = 0
  for (arg in args) {
   val num = arg.toIntOrNull()
   if (num == null) {
    call.respondText("Invalid arguments")
    return@get
   }
   sum += num
  }
  call.respondText("$sum")
 }
}
```

Similarly, you can use `request.headers.get()` and `reqiest.headers.getAll()` to obtain values of the request header data.

HTML DSL

The HTML DSL library together with the Ktor HTML builder allows you to generate responses based on the HTML content. This gives an alternative to techniques like JSP which embed the executable code into the UI markup. With HTML DSL, you have both a concise syntax and all the benefits of a Kotlin code, including type safety and the powerful IDE code insight. In this section, we won't be discussing any details of the DSL library and will simply focus on an example demonstrating its usage for building HTML forms. For more information, you can visit the HTML DSL website at **https://github.com/Kotlin/kotlinx.html**.

Let's create a simple web application for generating random numbers. Our server will present a page with a form where the user can enter the desired range and a number of values to generate as shown on Figure 16.12. The server will also perform a basic validation of input data ensuring that:

- All values are valid integers
- `From` bound is less than or equal to `To bound`

- `How many` field contains a positive number

When some of the requirements above are violated on form submission, the server will return the form back to the client with an error message(s) next to the corresponding text field(s).

The full text of the server module is rather long, so we won't put it here but focus on some specific passages instead. An interested readers can find it in the `ch16/random-gen` directory of the book supplementary repository available at **GitHub: https://github.com/asedunov/kotlin-in-depth**.

We've already seen an example of using HTML DSL for rendering simple elements like lists and paragraphs. So now, let's look at how to use it for building HTML forms. Consider the following code taken from the application module sources:

```
body {
 h1 { +"Generate random numbers" }
 form(action = "/", method = FormMethod.get) {
  p { +"From: " }
  p {
   numberInput(name = FROM_KEY) {
    value = from?.toString() ?: "1"
   }
   appendError(FROM_KEY)
  }
  p { +"To: " }
  p {
   numberInput(name = TO_KEY) {
    value = to?.toString() ?: "100"
   }
   appendError(TO_KEY)
  }
  p { +"How many: " }
  p {
   numberInput(name = COUNT_KEY) {
    value = count?.toString() ?: "10"
   }
   appendError(COUNT_KEY)
  }
  p { hiddenInput(name = GENERATE_KEY) { value = "" } }
  p { submitInput { value = "Generate" } }
 }
 …
}
```

Note the `form()` block inside the `body()`: This call defines an HTML form and introduces a scope where you can add input components such as text fields and buttons. The action argument specifies the target URL where the form data is sent to.

HTML DSL provides a whole set of functions for creating all basic kinds of input components such as:

- `input()`: A general-purpose text field
- `passwordInput()`: A text field for entering passwords
- `numberInput()`: A text field for numeric values with next/prior buttons
- `dateInput()/timeInput()/dateTimeInput()`: Specialized text fields for entering date and time
- `fileInput()`: A text field with a browse button for uploading a local file, and so on

The `submitInput()` call creates a Submit button which packs the form data into the HTTP request and sends it to the server.

If you look at the source of the page rendered in a browser, you'll be presented a markup similar to the following code:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Random number generator</title>
 </head>
 <body>
  <h1>Generate random numbers</h1>
  <form action="/" method="get">
   <p>From: </p>
   <p><input type="number" name="from" value="200"></p>
   <p>To: </p>
   <p>
    <input type="number" name="to" value="100">
    <strong> 'To' may not be less than 'From'</strong>
   </p>
   <p>How many: </p>
   <p>
    <input type="number" name="count" value="-10">
    <strong> A positive integer is expected</strong>
   </p>
   <p><input type="hidden" name="generate" value=""></p>
   <p><input type="submit" value="Generate"></p>
```

```
  </form>
 </body>
</html>
```

It’s not hard to see a direct correspondence between HTML tags and DSL blocks in the server code. Just like with the Anko layout, we can easily refactor and reuse this UI code which would’ve been noticeable if we’d decided to keep it as an HTML file resorting to JSP or some template engine-like Velocity to provide dynamic content. Figure 16.12 shows a rendering result after submitting the form with invalid data:

Figure 16.12: *Showing error messages*

Figure 16.13 shows a case of valid data with generated numbers added at the bottom of the page:

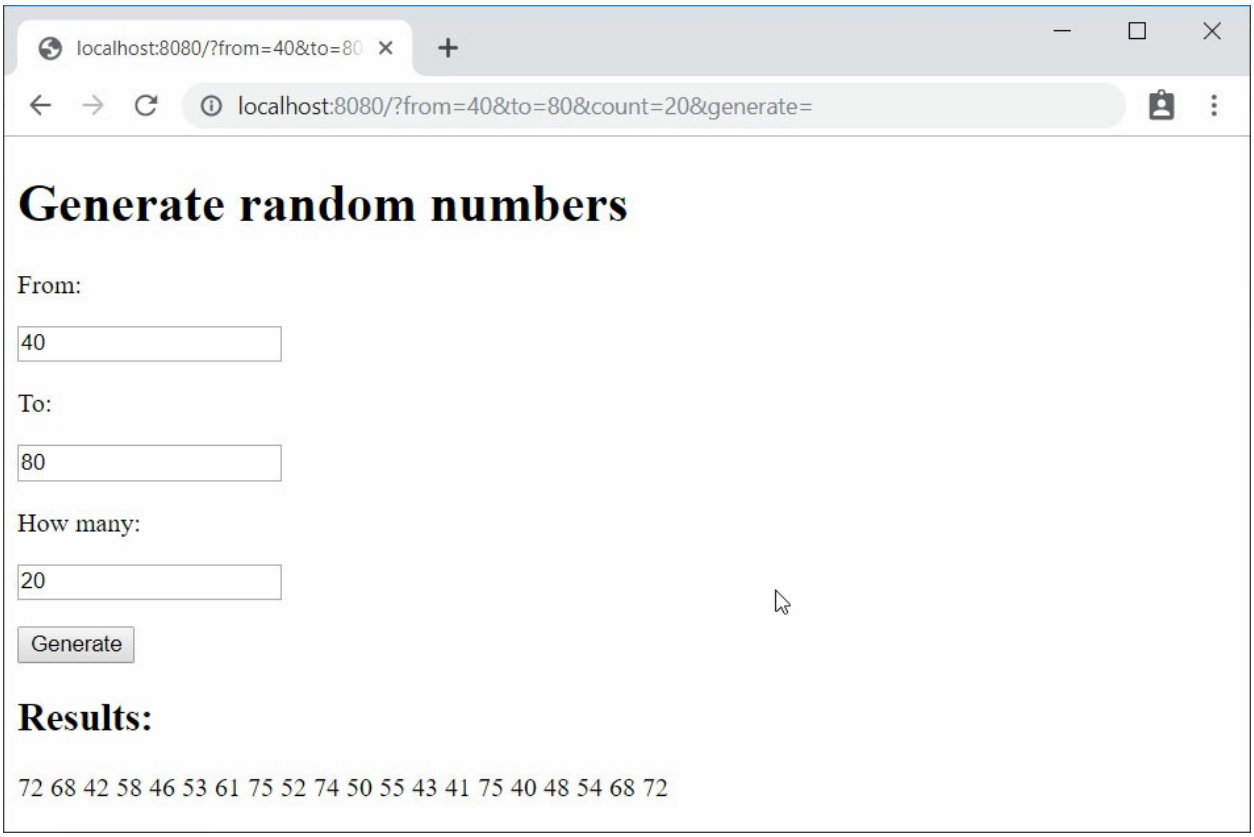

Figure 16.13: *Form with generation results*

The HTML DSL can also be used separately from the Ktor HTML builder library. In this case, you'll need to include a dependency on the DSL artifact itself. In Gradle, for example, this amounts to adding the following line into the corresponding dependencies block:

```
compile "org.jetbrains.kotlinx:kotlinx-html-jvm:0.6.12"
```

If your project doesn't use an external build system like Gradle or Maven, you can add the HTML DSL support by configuring a new library in the **`Project Structure`** dialog (similar to how we've done it with the Coroutines library in *Chapter 13, Concurrency*).

Apart from HTML, Ktor supports some popular template engines such as Velocity, Thymeleaf, and Mustache. You can find detailed information and examples on the Ktor website.

Sessions support

Ktor comes with a built-in support of session mechanism which allows web

applications to preserve some data between different HTTP requests and identify a particular client or user. User preferences, shopping cart items, and authorization data are common cases of information which can be kept in a session.

To use a session, you need to install a corresponding feature and specify how you want to store its data. For example, to keep the session inside a client cookie, you will write the following code:

```
install(Sessions) {
 cookie<MyData>(“my_data”)
}
```

The `MyData` here is a class which represents a session data; its instances can be accessed on the server-side using `ApplicationCall` and automatically serialized/deserialized when communicating with the client. The default serializer can handle classes with properties of simple types like Int or String, but you can override it by creating your own implementation of `SessionSerializer` and providing it in the `install()` block. The `my_data` value serves as a cookie key which distinguishes `MyData` instances from other sessions installed in the server.

Let’s consider an example which renders a simple HTML page and can track the number of times it’s been visited by a particular client. First, we’ll need a class to hold the counter:

```
data class Stat(valviewCount: Int)
private const val STAT_KEY = “STAT”
```

The `STAT_KEY` will be used as a key to distinguish the Stat instance among other session data.

Now, we can access the counter by using the `get()/set()/getOrSet()` methods of the `call.sessions` object. The following handler renders the message telling how many times the user has accessed the root page:

```
private suspend fun ApplicationCall.rootPage() {
 val stat = sessions.getOrSet { Stat(0) }
 sessions.set(stat.copy(viewCount = stat.viewCount + 1))
 respondHtml {
  body {
   h2 { +”You have viewed this page ${stat.viewCount} time(s)” }
   a(“/clearStat”) { +”Clear statistics” }
  }
 }
```

```
}
```

It’s reasonable to keep session instances immutable because the server usually runs in a multi-threaded environment, so keeping session states in mutable objects can lead to error-prone code. Instead, you read and replace the session as a whole using the `get()/set()` functions. The `getOrSet()` function allows you to initialize the session if it doesn’t yet exist.

The final step is to define the routine rule for both the root and `/clearStat` paths. In the latter case, we simply discard the session data and redirect the user back to the root page (which will display zero count at that point):

```
@Suppress(“unused”) // Referenced in application.conf
fun Application.module() {
 install(Sessions) {
  cookie<Stat>(STAT_KEY)
 }
 routing {
  get(“/”) {
   call.rootPage()
  }
  get(“/clearStat”) {
   call.sessions.clear(STAT_KEY)
   call.respondRedirect(“/”)
  }
 }
}
```

If you run the application and open `localhost:8080` in your browser, you’ll see that the view count increases each time the page is refreshed. Figure 16.14 shows results of four updates in a row:

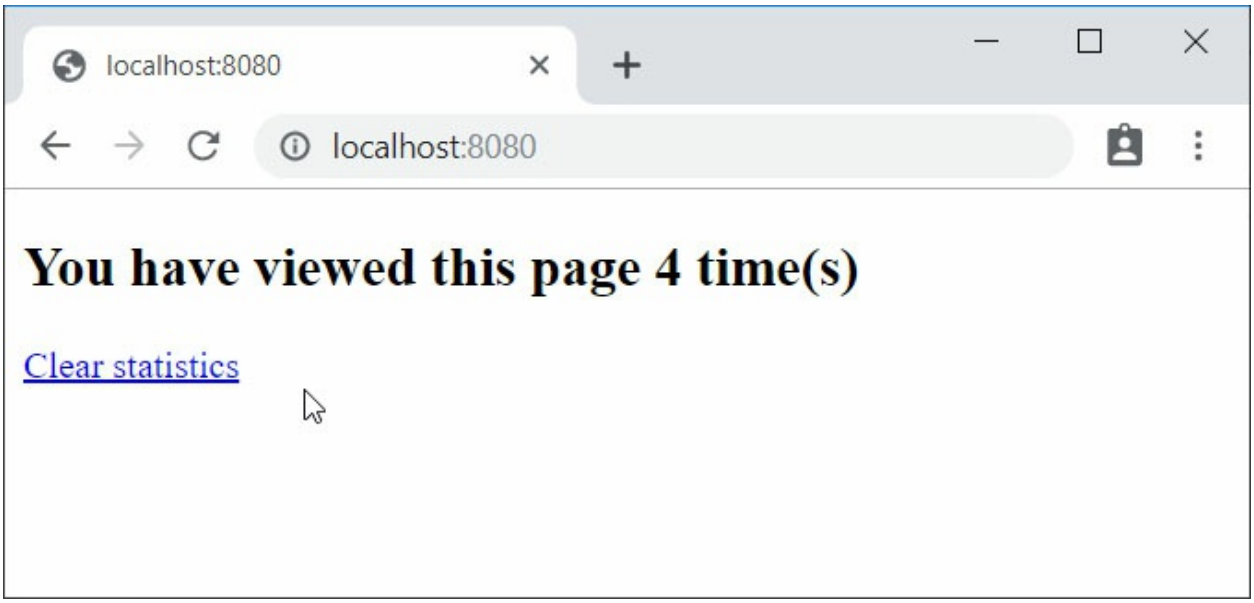

Figure 16.14: *Using session data to track page view count*

Clicking on the **`Clear statistics`** link forces the server to remove the session data and reset the count back to zero. Note the `responseRedirect()` call at the end of the /`clearStat` handler; it's needed to render the page again after the session is cleared.

As an alternative to cookies, you may also store sessions in the header of HTTP requests and responses:

```
install(Sessions) {
 header<MyData>("my_data")
}
```

Ktor sessions may be stored at either client or server side. By default, all the session data are transferred to the client who keeps them and sends back with a next request. This may pose a security issue since a built-in serializer represents the session data as a plain text. To overcome this problem, Ktor provides a session transformer mechanism which implements additional encoding/decoding of transferred data.

One of the built-in transformers, `SessionTransportTransformerMessage Authentication` accompanies the session data with their hash computed according to a specific algorithm (SHA256 by default). In a simplest case, you just need to provide a secret key:

```
install(Sessions) {
 cookie<Stat>(STAT_KEY, SessionStorageMemory()) {
  val key = Random.Default.nextBytes(16)
```

```
    transform(SessionTransportTransformerMessageAuthentication(key)
  }
}
```

The original session data remain unchanged, so the third party can still view them on the client-side. They, however, won't be able to change the session data without the server's consent because that would invalidate the digest and computing a new one relies on the knowledge of a secret key.

A stronger security guarantee is given by `SessionTransportTransformerEncrypt` which encrypts the session data preventing their read by a third party. To configure this transformer, you'll need to provide both an encryption and authentication key (the latter is used to create a digital signature of the session data):

```
install(Sessions) {
  cookie<Stat>(STAT_KEY) {
    val encryptionKey = Random.Default.nextBytes(16)
    val signKey = Random.Default.nextBytes(16)
    transform(
      SessionTransportTransformerEncrypt(encryptionKey, signKey)
    )
  }
}
```

It's also possible to add your own transformer by implementing the `SessionTransportTransformer` interface.

By default, both the `cookie()` and `header()` blocks configure client-side sessions; in this case, all the session data are stored on the client-side and transferred to/from the server with each request/response. Alternatively, you can configure a session storage which tells Ktor to store the session body on the server-side and transfer only session IDs:

```
install(Sessions) {
  cookie<Stat>(STAT_KEY, SessionStorageMemory())
}
```

`SessionStorageMemory` is a built-in implementation which keeps the session data in a server memory. Note that memory consumption grows with a number of active clients so server-side sessions are worth being kept as compact as possible.

This concludes an overview of basic features available to server applications of Ktor. In the next section, we'll focus on the other side of communication

and look at using Ktor for programming HTTP clients.

Client features

Ktor is not limited to writing server applications and can be used to greatly simplify development of asynchronous clients communicating with various services. In this section, we'll focus on small subset of its features centered on the `HttpClient` class which allows you to communicate with web servers using the HTTP protocol.

Requests and responses

The basic functionality of any HTTP client revolves around composing requests for some web servers and reading its subsequent responses. Ktor provides a rich set of out-of-the-box primitives for working with both text and binary content you might see in HTTP requests and responses.

The simplest way to issue an HTTP request via `HttpClient` is to use its generic `get()` method and pass a target URL. The method type arguments determine what kind of object is returned by the client to represent a server response. For example, to obtain a response body as a single piece of text, you may use `get<String>()`:

```
import io.ktor.client.HttpClient
import io.ktor.client.request.get
import kotlinx.coroutines.runBlocking

enum class DayOfWeek {
 SUNDAY,
 MONDAY,
 TUESDAY,
 WEDNESDAY,
 THURSDAY,
 FRIDAY,
 SATURDAY
}

fun main() {
 runBlocking {
  HttpClient().use {
   val url =
   "http://worldtimeapi.org/api/timezone/Europe/London.txt"
   val result = it.get<String>(url)
   val prefix = "day_of_week:"
```

```
    val from = result.indexOf(prefix)
    if (from < 0) return@runBlocking
    val to = result.indexOf('\n', from + 1)
    if (to < 0) return@runBlocking
    val dow = result
      .substring(from + prefix.length, to)
      .trim()
      .toInt()
    println("It's ${DayOfWeek.values().getOrNull(dow)} in
    London!")
   }
  }
}
```

Apart from string representation, we may access the response body using the binary form by converting it to a byte array:

```
val bytes = client.get<ByteArray>(url)
```

Or obtaining an asynchronous `ByteReadChannel`:

```
val channel = client.get<ByteReadChannel>(url)
```

Request-making methods of `HttpClient` are suspending functions and thus must be called in some coroutine context. That's why we've used the `runBlocking()` in the examples above. In general, you're free to use any asynchronous computation primitives offered by the Kotlin coroutines.

Note that `HttpClient` requires explicit finalization via the `close()` method. When its scope is limited, we can hide its call behind the `use()` block just like how we do it with any other instance of the `Closeable` type.

As you've probably guessed, the `get()` method directly corresponds to the HTTP `GET`. Ktor client provides similar shorthands for all methods supported by the HTTP 1.x/2.x standard: `post()`, `put()`, `delete()`, `patch()`, `head()`, `options()`.

These methods accept an optional lambda of `HttpRequestBuilder.() -> Unit` type where you can configure additional request parameters such as adding headers or body. To add a header, you can use the headers method defined in `HttpRequestBuilder`:

```
client.get<ByteArray>(url) {
 header("Cache-Control", "no-cache")
}
```

It’s also possible to use HeadersBuilder available via the headers property or its namesake block:

```
client.get<ByteArray>(url) {
 headers {
  clear()
  append(“Cache-Control”, “no-cache”)
  append(“My-Header”, “My-Value”)
 }
}
```

`HttpClient` provides a simplified way to supply the User-Agent header which allows the server to identify client software (such as a web browser and its particular version). To do this, you just need to install the `UserAgent` feature and specify the header value using the agent property:

```
val client = HttpClient(Apache) {
 install(UserAgent) {
  agent = “Test Browser”
 }
}
```

You can also use one of the predefined User-Agent settings:

- `BrowserUserAgent()`: Includes popular browsers like Chrome or Safari
- `CurlUserAgent()`: Correspond to the Curl agent

The preceding functions replace the entire feature installation block. For example:

```
val client = HttpClient() {
 BrowserUserAgent()
}
```

To supply a request body (e.g. for a `POST` request), you can use the body property of `HttpRequestBuilder`. The simplest case is writing a `String` representation:

```
client.get<String>(url) {
 body = “my_key1=my_value1&my_key2=my_value2”
}
```

Alternatively, you can supply any implementation of `OutgoingContent` such as `TextContent` which is similar to writing a `String` but additionally allows you to specify a MIME type, `ByteArrayContent` which is useful to pass

binary data, `LocalFileContent` which allows you to transfer a file, and so on. Additionally, by installing the `JsonFeature`, you can enable automatic serialization of arbitrary objects in the JSON form.

The `submitForm()` function implements a common scenario by imitating the behavior of HTML forms. For example, the following code submits the form data for the server application we've demonstrated in the **`HTML DSL`** section:

```
val result = client.submitForm<String>(
 url = "http://localhost:8080",
 encodeInQuery = true,
 formParameters = parametersOf(
  "from" to listOf("0"),
  "to" to listOf("100"),
  "count" to listOf("10"),
  "generate" to emptyList()
 )
)
```

The parameters are passed as a set of key-value pairs while the `encodeInQuery` argument determines their representation as a part of request data:

- `true`: HTTP `GET` with parameters encoded in the request URL
- `false`: HTTP `POST` where parameters passed in the request body

The Ktor client comes with an out-of-the-box support of HTTP redirects. This feature is installed by default, so whenever the server sends back a response with redirect status, the client automatically follows a new location.

Cookies

HTTP cookies provide a common way to preserve some state between client requests by passing around small packets of data inside HTTP headers. The client (such as a web browser) initially obtains a cookie from the server and stores in on the client-side adding to subsequence requests. Cookies are particularly useful for maintaining server sessions, a feature which we've already covered in the previous sections. Now, we'll look at how cookies can be used on the client side.

If an HTTP server uses cookies to preserve some data between client calls, the client has to arrange proper storage of such data and provide them when

necessary with HTTP requests. Ktor simplifies this task by providing ready-to-use cookies feature. To demonstrate its usage, let's write a simple code for the view counter application we've discussed in the server section (see Figure 16.14):

```
package com.example
import io.ktor.client.HttpClient
import io.ktor.client.engine.apache.Apache
import io.ktor.client.features.cookies.HttpCookies
import io.ktor.client.request.get
import kotlinx.coroutines.*
fun main() {
 HttpClient(Apache) {
  install(HttpCookies)
 }.use { client ->
  runBlocking {
   repeat (5) {
    val htmlText = client.get<String>("http://localhost:8080")
    val from = htmlText.indexOf("<h2>")
    val to = htmlText.indexOf("</h2>")
    if (from < 0 || to < 0) return@runBlocking
    val message = htmlText.substring(from + "<h2>".length, to)
    println(message)
    delay(500)
   }
  }
 }
}
```

As you can see, our client retrieves the root-path (/) response, finds a header enclosed inside a `<h2>` tag, and prints it to the standard output. Note the `install(HttpCookies)` call which configures `HttpClient` to handle cookies. Since the request/response cycle is repeated five times (each time with an updated cookie), the output will look like:

```
You have viewed this page 0 time(s)
You have viewed this page 1 time(s)
You have viewed this page 2 time(s)
You have viewed this page 3 time(s)
You have viewed this page 4 time(s)
```

By default, an HTTP client starts with empty cookies and uses data provided by the server passing them together with a next request. This corresponds to typical browser behavior. Sometimes, you may need to send a request with a preconfigured set of cookies without getting them from the server – say, to

use them in a test case which verifies a server response. In this case, you may change the cookies storage policy by changing the storage property to `ConstantCookiesStorage` and supplying a set of Cookie objects. The client will then ignore any new cookies sent back by the server and add the same data to each request. To see this feature in action, we'll need to run a plain-text version of our server without any cookies transformations. Now, change the client definition to the following:

```
val client = HttpClient(Apache) {
 install(HttpCookies) {
  storage = ConstantCookiesStorage(Cookie("STAT",
  "viewCount=%23i2"))
 }
}
```

It's not hard to see that this cookie forces the `viewCount` variable to take the value of 2. As a result, when we rebuild and run the client application, the server will simply repeat the same response five times:

```
You have viewed this page 2 time(s)
You have viewed this page 2 time(s)
You have viewed this page 2 time(s)
You have viewed this page 2 time(s)
You have viewed this page 2 time(s)
```

The default behavior where cookies are automatically taken from the server is given by the `AcceptAllCookiesStorage` class. On top of that, you can add your own storage policy by implementing the `CookiesStorage` interface.

Conclusion

This chapter has introduced us to the basic features of the Ktor framework aimed at creating connected client/server applications. We got an understanding of the basic Ktor project structure and its common features provided for both server-and client-side applications such as handling requests and responses, describing routing rules, and using sessions. The material here will help you to get a grip on basic ideas in preparation for a more thorough investigation what Ktor can offer to Java/Kotlin developers. We recommend you to start with an official Ktor site (**https://ktor.io**) and give special consideration to the **`Samples`** section at **https://ktor.io/samples**.

In the next chapter, we'll continue with the connectivity topic and talk about

using Kotlin for development of microservices. We'll discuss the basics of the microservice architecture and look at how Kotlin can help us in creating them on the platform of Ktorans Spring Boot.

Questions

1. Describe the basic steps of the Ktor project configuration.
2. How do you generate an HTML-based content in Ktor? Explain the basic features of HTML domain-specific language.
3. How do you extract client-supplied data from HTTP requests?
4. Explain the basic ways to generate an HTTP response in Ktor.
5. Describe Ktor routing DSL.
6. How can you add a session support to your web application? Explain differences between client and server sessions.
7. Describe how to build and send an HTTP request using Ktor.
8. How can you access the body and headers of an HTTP response using the Ktor client?
9. Describe the client-side use of cookies in Ktor.

CHAPTER 17

Building Microservices

The microservice architecture provides a way of building applications which consists of multiple interconnected components aimed at performing fine-grained domain-specific tasks. This architecture contrasts with a more traditional technique of creating a monolithic application which is deployed as a whole. Microservices facilitate modular development by allowing you to physically separate pieces of functionality and ease development, testing, and deployment/update of individual application parts.

In this chapter, we'll explain the basics of microservice architecture as well as its defining principles and look at how Kotlin can help you in microservice implementation on the example of Spring Boot and Ktor frameworks. The Spring framework is a commonly used tool in the Java world which has a special focus on the Kotlin support in its recent versions while Ktor, we've already discussed in the previous chapter, is specifically targeted at development of various types of connected applications and makes heavy use of Kotlin features. Having worked through the chapter, you will be able to compose simple services and have the necessary foundation for further learning of more specific microservice frameworks.

Structure

- The microservice architecture
- Introducing Spring Boot
- Microservices with Ktor

Objectives

To understand the fundamental principles of the microservice architecture and learn the basics of creating microservices with Spring Boot and Ktor frameworks.

The microservice architecture

The big idea of the microservice architecture is to replace a monolithic application – deployed and delivered as a whole – by a set of lightweight loosely-coupled services; each having a specific task and communicating with other services using well-defined protocols.

To give a more specific example, suppose that we want to build an online store-like application which provides users with basic set of features like browsing goods catalog and making orders. By following a monolithic application approach, we might come up with a design similar to the one shown in *Figure 17.1*:

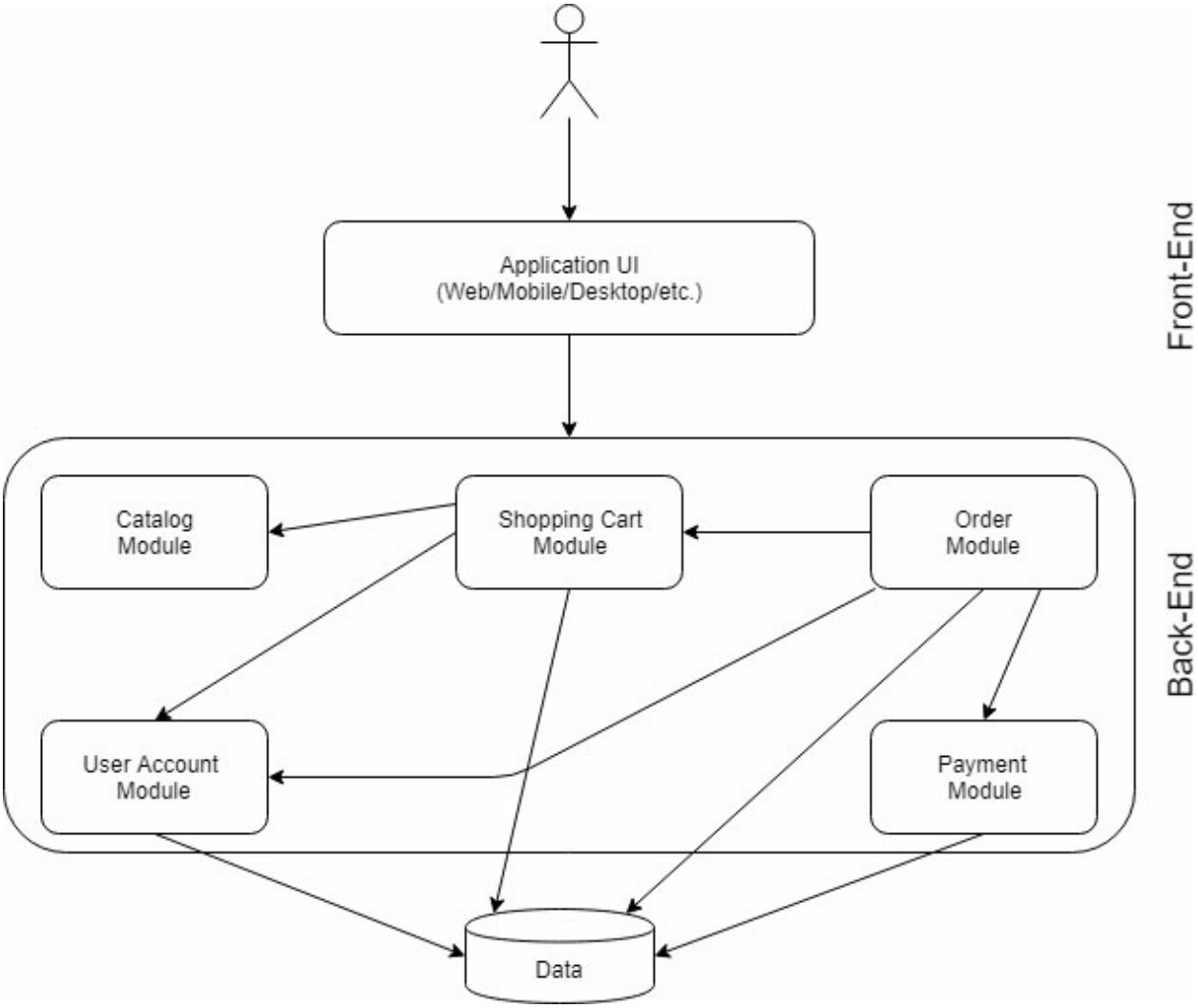

Figure 17.1: *A monolithic application example*

This is a common three-level architecture which includes separate layers for

the application UI (be it a desktop, web or mobile client), business logic, and data storage/retrieval. The back-end part of the application which is responsible for implementing its domain-specific workflows can be decomposed into more specific modules such as providing access to the catalog, maintaining user's shopping cart, placement/tracking/cancellation of orders, payments as well as authentication, and user profile management. Note that although modules themselves might be loosely-coupled, they are not usually distributed or deployed independently making the server application a monolith.

This approach, however, can pose certain problems as the application grows. Any change in the codebase be it an implementation of some new feature or a bug fix requires you to update/redeploy the entire application which increases its startup time and introduces an opportunity for new bugs. This also hinders the application scalability. With a monolithic approach, you have to deal with scaling the entire application which is significantly more complicated than scaling specific modules or functions. One more issue to consider is reliability since running all back-end modules under the same process makes your application more vulnerable to possible memory leaks and other kinds of bugs.

A Service-oriented architecture (SOA) mitigates these problems by means of decomposing a monolithic application into a set of self-contained services which can be developed, updated, and deployed largely independently. Microservices can be considered a step in the SOA evolution with the focus on making services as small and simple as possible; although in practice both terms are often used as synonyms.

If we try to break down our original monolithic application design, we might end up with something resembling *Figure 17.2*:

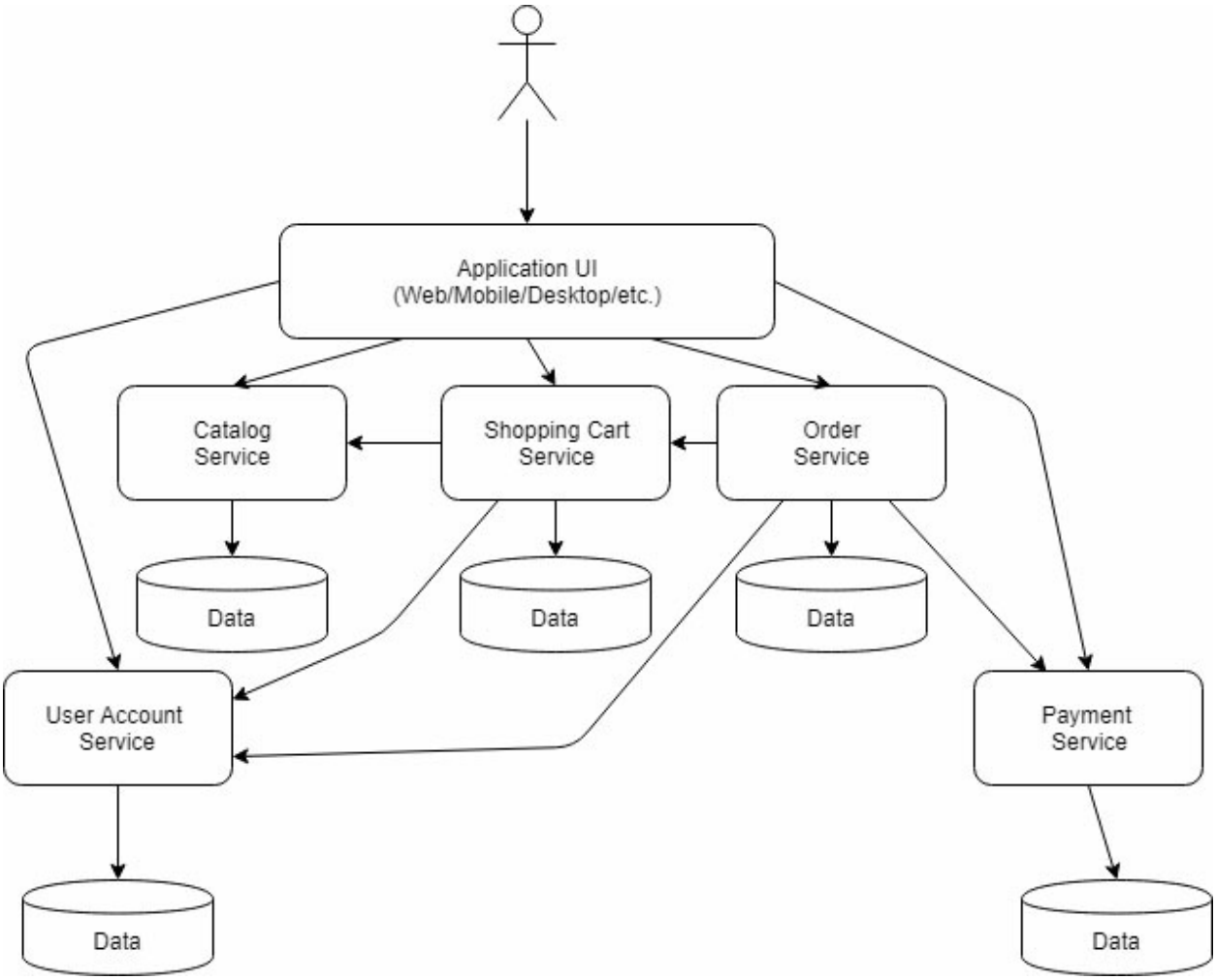

Figure 17.2: *A microservice architecture*

You can see that the original modules are replaced with services which perform the same functions and communicate with each other using some kind of network protocol like HTTP. Now, individual services can be developed, updated, and configured more or less independently from each other. They may also use separate databases which might even be managed by different DBMS.

Although microservices lack a rigorous definition, all their practical implementations are based on a common set of principles:

- Each microservice is focused on performing some domain-specific task such as managing goods catalog or user's shopping cart in our e-commerce application example.
- Microservices communicate using some well-defined protocol which

effectively establishes their API. A common case involves using HTTP combined with XML and JSON formats for transferring complex data as well as **RPC (remote procedure call)-**based protocols.

- Microservices can be independently versioned, deployed, and updated
- Microservices are language-and framework-agnostic, which means that, in general, you can implement them in any programming language you deem fit for the purpose and use any development framework of your choice. All that matters is a communication protocol your service will use for interacting with others.

This should give you a basic understanding of what a microservice architecture is and in what cases you might want to employ it in your application. Later, we'll demonstrate how a microservice programming may look in the context of the Kotlin language. It will provide a foundation for your own investigation of more specific technological stacks and frameworks such as Spring, Netflix, or Ktor.

Introducing Spring Boot

Spring is one of the most frequently used Java framework which provides a rich set of facilities for building various applications with a primary focus on the J2EE platform. In this chapter, we'll talk about using a powerful Spring/Kotlin combination for development of microservices on the example of a Spring Boot project. In general, Spring Boot is a collection of utilities simplifying the setting up of various Spring project types and the framework configuration. Similarly, we'll start with guiding you through basic steps required for creating a Spring Boot microservice.

Setting up a project

One of the easiest ways of starting a Spring application is to use a special web tool called Spring Initializr to automatically generate a project skeleton based on the chosen application type. To make use of this tool, open **https://start.spring.io in** your browser (see *Figure 17.3*):

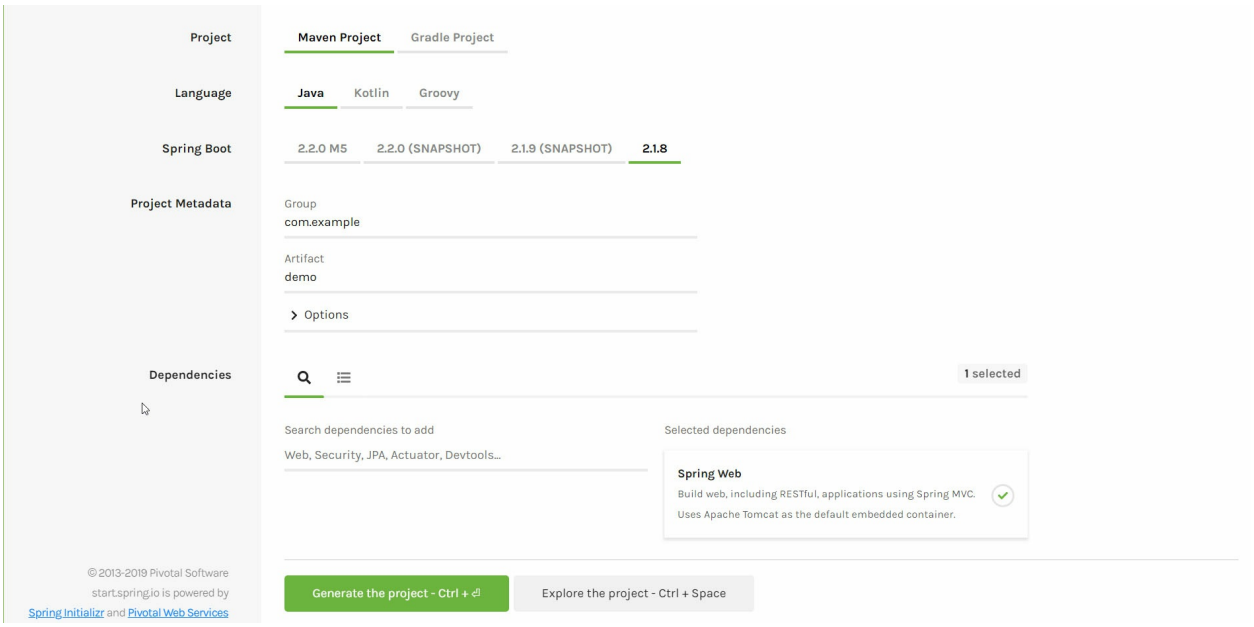

Figure 17.3: *Using Spring Initializr to generate a new project*

This page allows us to choose a set of basic options which determine the type of project the Initializr will generate:

- A build system type (Maven/Gradle) that will be used to configure and build a project from sources; for our example, we'll use Gradle as it gives you a more flexible and concise way to adjust the project configuration.
- The primary language of a new project which initializer will use to generate a sample source code (Kotlin in our case). This will also affect the project configuration as using Kotlin, for example, will require some additional dependencies in Maven/Gradle build files.
- Version of Spring Boot to use: We'll choose the latest release version of Spring at the time of writing the book, which is 2.1.8.
- Project group and artifact ID which define its Maven coordinates for artifact publication.

Additionally, you may use the Dependencies field to specify some common packages to be included into our project. Since our services will use HTTP, we'll need the web support. Type Web in the field and choose the **`Spring Web`** option in the suggestion list.

After choosing all the necessary options, click on the **`Generate the project`** button and download a ZIP file containing an initializer-created project. To

open the project in an IntelliJ IDEA, you need to do the following:

1. Unpack the archive to some local directory.
2. Call the **`File | New | Project`** from **`Existing Source…`** menu command and specify the path to the unpacked project root as well as a build system kind (Gradle).
3. Wait until the IDE finishes its synchronization with the Gradle build model after which you'll see a project structure similar to the one in *Figure 17.4*:

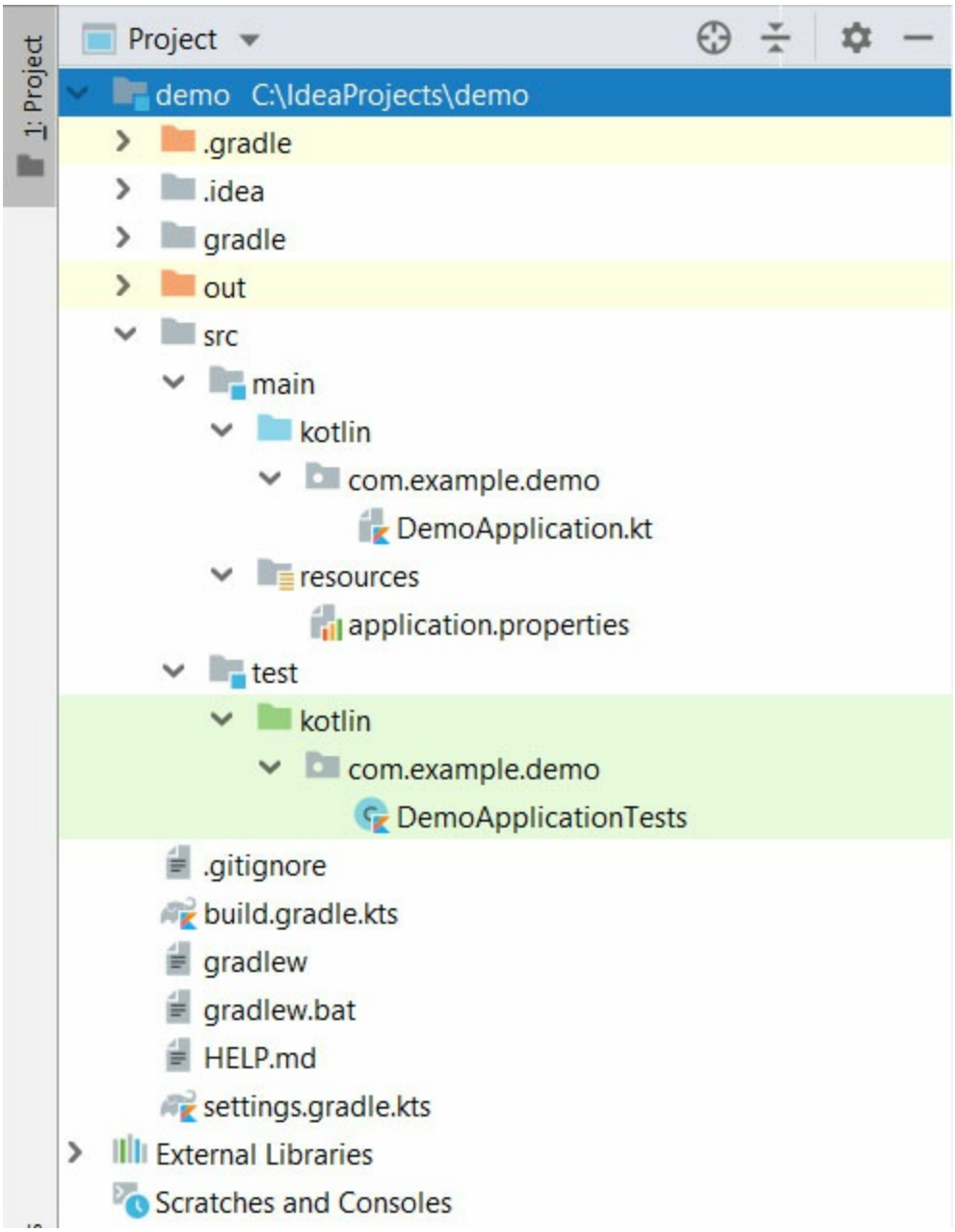

Figure 17.4: *Structure of a sample Spring Boot project*

Similarly, in a project generated by the Ktor wizard, the `build.gradle.kts` file will contain definition of project dependencies. Note down the `.kts` extension. It means that the buildfile is written in Kotlin rather than Groovy. For this reason, the script follows a slightly different syntax than the ones we've seen in Ktor and Android examples:

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
 id("org.springframework.boot") version "2.1.7.RELEASE"
 id("io.spring.dependency-management") version "1.0.8.RELEASE"
 kotlin("jvm") version "1.3.41"
 kotlin("plugin.spring") version "1.3.41"
}

group = "com.example"
version = "0.0.1-SNAPSHOT"
java.sourceCompatibility = JavaVersion.VERSION_1_8

repositories {
 mavenCentral()
}

dependencies {
 implementation("org.springframework.boot:spring-boot-starter-
 web")
 implementation("com.fasterxml.jackson.module:jackson-module-
 kotlin")
 implementation("org.jetbrains.kotlin:kotlin-reflect")
 implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
 testImplementation(
  "org.springframework.boot:spring-boot-starter-test"
 )
}

tasks.withType<KotlinCompile> {
 kotlinOptions {
  freeCompilerArgs = listOf("-Xjsr305=strict")
  jvmTarget = "1.8"
 }
}
```

You might want to make some adjustments such as upgrading the Kotlin version to a more recent one before proceeding. Just like with any other Gradle project, IDEA will suggest you to resynchronize a project model upon any change in the `build.gradle` file (unless you have autoimport switched

on, in which case synchronization starts up automatically).

The `application.properties` file contains various properties (in a simple `key=value` format) affecting the Spring behavior. By default, it's empty but later, we'll use it to change the port our service will listen.

The entry point of our project is defined in the `DemoApplication.kt` file which contains the definition of the `DemoApplication` class and the `main()` function delegating the application start-up to the framework:

```
package com.example.demo

import
org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
 runApplication<DemoApplication>(*args)
}
```

The `runApplication()` function will create the `DemoApplication` instance as well as automatically instantiate and wire all services required for our application. For a web application with a default configuration like ours, it also starts a bundled Tomcat server which is going to handle client requests dispatching them to the Spring-supplied servlet. The `DemoApplication` instance will serve as a global context which can be injected into other application components when necessary. Note the `@SpringBootApplication` annotation; this is handy shortcut which allows you to configure a given class as a Spring application context.

After the application is started (you use the **`Run`** command from IDEA main menu to do this), we can access it using an HTTP client such as a web browser. Since our application doesn't contain the actual request processing code yet, the Spring servlet will response with a standard error page on every request we make. *Figure 17.5* shows you an example:

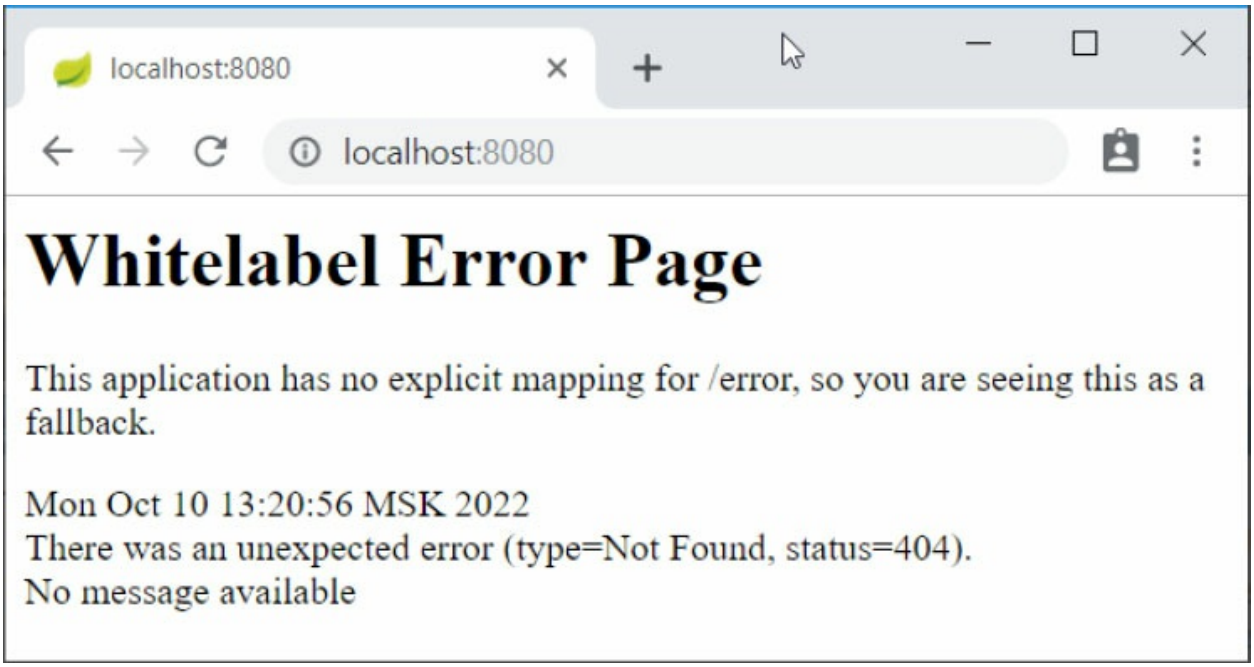

Figure 17.5: *Default response page provided by the Spring framework*

Note that Spring uses `8080` port to listen to the client requests unless it's explicitly changed in the `application.properties` file.

> **IDE Tips: There are plugins which add the Spring support to IntelliJ IDEA and, in particular, allow one to generate various Spring-powered projects similarly to the Initializr tool. Note, however, that these plugins are not available in the IDEA Community Edition. IDEA Ultimate, on the other hand, has them bundled out of the box.**

In the upcoming sections, we'll use this project stub as a base for creating sample microservices. But the first thing we have to do before getting to the actual coding is to define what our services will do and how they will communicate with their clients. To demonstrate a common practice of implement microservices as small web applications, we'll use HTTP as a base of our example communication protocol.

Deciding on the Services API

In this chapter, we'll walk you through a simple example of designing a pair of communicating services. The first service will be similar to a random number generator we've demonstrated in *Chapter 16, Web Development with Ktor*, but will have a more formalized input and output to be usable in the

form of API. When given a request with the URL of the form:

`/random/int/from/X/to/Y/quantity/N`

It will produce a list of N random numbers in the range between `X` and `Y` (both inclusive). The result will be given as a JSON object with a pair of fields:

- `status`: A string which contains error message or null in case of successful completion.
- `values`: An array of generated integers (empty when the status signifies an error).

Possible cases of error status:

- Non-integer values for `X`, `Y` or `N`
- Non-positive `n`
- `Y<X`

Let's give some examples of the expected service output for a given URL:

URL example	Service response
`/random/int/from/10/to/20/quantity/5`	`{"status":null, "values":[16,17,18,17,12]}`
`/random/int/from/20/to/10/quantity/5`	`{"status":"Range may not be empty", "values": []}`
`/random/int/from/10/to/20/quantity/-1`	`{"status":"Quantity must be positive", "values":[]}`
`/random/int/from/1X/to/20/quantity/5`	`{"status":"Range start must be an integer", "values":[]}`

Table 17.1: *Examples of number generator output*

Another function of our service will be generation of floating-point numbers. We'll use a URL of the following form:

`/random/float/quantity/N`

This URL will get the service to produce N double precision numbers in the range between 0 and 1 (excluding 1).

The second service will provide a similar API for generating random passwords. Given a URL of the form:

`/password/length/L/quantity/N`

It will produce N alphanumeric strings each having length L. The password generator will use the same output format; the only difference being that the values field will be an array of strings rather than numbers.

URL example	Service response
`/password/length/8/quantity/5`	`{"status":null, “values”: [“B0zDWtvG”,”JrSkXl7X”, “oDwR7cp2”,”X8sRfzDW”,”nUcRXzn1”]}`
`/password/length/bbb/quantity/5`	`{"status": “Length must be an integer”, “values”:[]}`
`/password/length/-1/quantity/ccc`	`{"status": “Length must be positive”, “values”:[]}`
`/password/length/8/quantity/-5`	`{"status": “Quantity must be positive”,”values”:[]}`

Table 17.2: *Examples of password generator output*

To demonstrate service communication, we’ll make a password generator which depends on the numeric one. When requested for a new password(s), it will call the number generator first to produce a series of random indices which are then turned to characters and joined together to produce strings.

Now that it’s clear how our service API will look, we can get to the actual implementation. We’ll start with the random number generator since it’s going to be used by another service.

Implementing a random generator service

Let’s set up a new Spring Boot project for our generator service following the steps from the *Setting up a project* section. The service entry point will become largely unchanged. In our case, it’ll be enough to rename the application class and package:

```
package com.example.randomGen

import
org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class RandomGenerator

fun main(args: Array<String>) {
 runApplication<RandomGenerator>(*args)
```

```
}
```

Before writing the actual business logic of the service itself, we need to define classes which hold up the data we're going to pass around in a JSON form. Since our service input consists of primitive values passed in the URL path, the only structured data is its output. That's exactly the job for a Kotlin data class:

```
package com.example.randomGen

data class GeneratorResult<T>(
 val status: String?,
 val values: List<T>
)

fun <T>errorResult(status: String) =
 GeneratorResult<T>(status, emptyList())
fun <T>successResult(values: List<T>) =
 GeneratorResult<T>(null, values)
```

The pair of utility functions, `errorResult()` and `successResult()`, will come in handy to simplify the construction of `GeneratorResult` in the service code.

The core logic of service is implemented in the so-called controller class which handles processing of client requests. To turn a given class into the Spring controller, you just need to annotate it with `@RestController`. The Spring will automatically load the class and create its instance during component scanning. We won't discuss the scan process in detail here, but you can find them in the Spring framework documentation (see, for example, the `@ComponentScan` annotation).

The stub of our controller class will therefore look like this:

```
package com.example.randomGen

import org.springframework.web.bind.annotation.*

@RestController
class RandomGeneratorController
```

To define a request handler, we mark the controller's methods with special annotation which associate them with specific request attributes. For example, the `@RequestMapping` annotation allows you to bind a method to requests with a particular URL:

```
@RequestMapping("/hello")
```

```
fun hello() = "Hello, World"
```

Similarly, in Ktor, you can use wildcards like * and parameter names to define path templates. In the following example, the last portion of the URL path gets automatically bound to the method parameter marked with the `@PathVariable` annotation:

```
@RequestMapping("/hello/{user}")
fun hello(@PathVariable user: String) = "Hello, $user"
```

Method parameter names may differ from variables you use in the path template; in this case, you need to specify the path parameter as the `@PathVariable` argument:

```
@RequestMapping("/sum/{op1}/{op2}")
fun hello(
 @PathVariable("op1") op1Str: String,
 @PathVariable("op2") op2Str: String
): Any {
 val op1 = op1Str.toIntOrNull() ?: return "Invalid input"
 val op2 = op2Str.toIntOrNull() ?: return "Invalid input"
 return op1 + op2
}
```

Apart from the URL path, the `@RequestMapping` annotation allows you to associate handlers based on various request data such as the HTTP method (`GET`, `POST`, etc.), content of headers and request parameters. Similarly, there are some alternatives to `@PathVariable` you can use to bind method parameters to request parameters (`@RequestParam`), request headers (`@RequestHeader`), session data (`@SessionAttributes`), and so on. The mapping options are quite similar to the routing mechanism of the Ktor; although in the case of Ktor, it’s specified as a piece of ordinary Kotlin code rather than some metadata in an annotation form. We won’t delve into the details here but interested readers can find relevant documentation on the Spring site at `docs.spring.io`.

When several methods of the controller share a common path prefix, it may be convenient to add `@RequestMapping` to the controller class as well. In this case, paths mentioned in the method-level annotations are defined relative to the class one. For example, instead of writing:

```
@RestController
class SampleController {
 @RequestMapping("/say/hello/{user}")
```

```
 fun hello(@PathVariable user: String) = "Hello, $user"

 @RequestMapping("/say/goodbye/{user}")
 fun goodbye(@PathVariable user: String) = "Goodbye, $user"
}
```

We can extract the common `/say` part into the `SampleConroller's` annotation:

```
@RestController
@RequestMapping("/say")
class RandomGeneratorController {
 @RequestMapping("hello/{user}")
 fun hello(@PathVariable user: String) = "Hello, $user"
 @RequestMapping("goodbye/{user}")
 fun goodbye(@PathVariable user: String) = "Goodbye, $user"
}
```

Having this in mind, let's implement the controller method which is going to take care of `/random/int` paths according to our service API:

```
@RequestMapping("/int/from/{from}/to/{to}/quantity/{quantity}")
fun genIntegers(
 @PathVariable("from") fromStr: String,
 @PathVariable("to") toStr: String,
 @PathVariable("quantity") quantityStr: String
): GeneratorResult<Int> {
 val from = fromStr.toIntOrNull()
  ?: return errorResult("Range start must be an integer")
 val to = toStr.toIntOrNull()
  ?: return errorResult("Range end must be an integer")
 val quantity = quantityStr.toIntOrNull()
  ?: return errorResult("Quantity must be an integer")
 if (quantity <= 0) return errorResult("Quantity must be
 positive")
 if (from > to) return errorResult("Range may not be empty")
 val values = (1..quantity).map { Random.nextInt(from, to + 1) }
 return successResult(values)
}
```

Handling of floating-point numbers corresponding to the /random/float paths can be done in a similar way. The full source text of this service can be found at **https://github.com/asedunov/kotlin-in-depth/ch17/number-gen-service**.

If we start our application and try to access the service via a browser, we'll get an expected response. You can see an example of getting a list of random numbers in *Figure 17.6*:

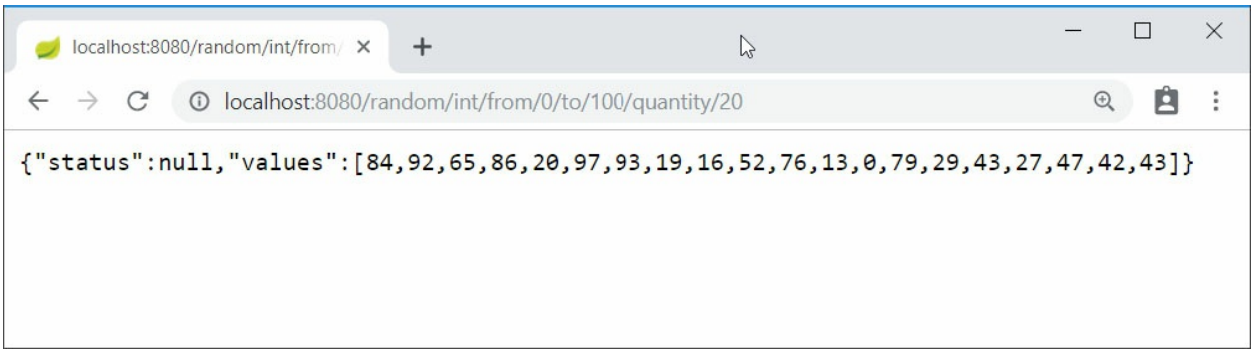

Figure 17.6: *An example of success response*

You can also make sure that our service can correctly handle common errors in client requests. For example, Figure 17.7 shows the result you get when requesting numbers in the range from 50 to 20:

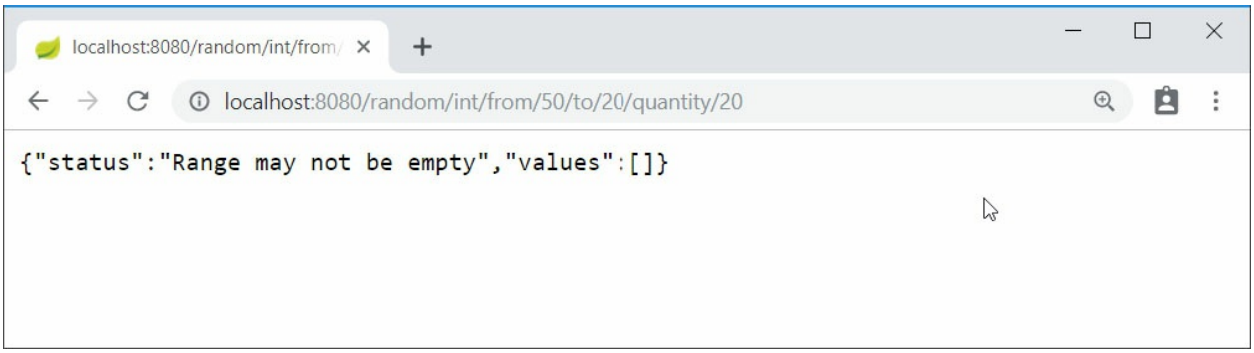

Figure 17.7: *Generator responding with an error*

As expected, the service responds with an error indicating that the specified interval is empty since its upper bound is less than the lower one.

Implementing a password generator service

Now, we can easily implement a second password-generating service using first as the starting point. Let's create a similar Spring Boot project placing our code into the `com.example.passwordGen` package.

The crucial difference from the random number generator is that the second service will have to communicate with the first one. Spring comes with out-of-the-box `RestTemplate` class which simplifies making requests to other web applications and retrieves their responses. For example, the code:

```
val url =
"http://localhost:8080/random/int/from/0/to/10/quantity/5"
val restTemplate = RestTemplate()
```

```
val result = restTemplate.getForObject(url,
GeneratorResult::class.java)
 as GeneratorResult<Int>
```

will return a result containing a list of five random integers in the range between 0 and 10.

Let’s now use this idea to transform numbers into password characters. Here is the full text of password generator controller class:

```
package com.example.passwordGen

import org.springframework.web.bind.annotation.*
import org.springframework.web.client.RestTemplate

private val chars = ('a'..'z') + ('A'..'Z') + ('0'..'9')
@Suppress("unused")
@RestController
@RequestMapping("/password")
class PasswordGeneratorController {
 @RequestMapping("/length/{length}/quantity/{quantity}")
 fun genPasswords(
 @PathVariable("length") lengthStr: String,
  @PathVariable("quantity") quantityStr: String
 ): GeneratorResult<String> {
  val length = lengthStr.toIntOrNull()
   ?: return errorResult("Length must be an integer")
  val quantity = quantityStr.toIntOrNull()
   ?: return errorResult("Quantity must be an integer")
  if (quantity <= 0) return errorResult("Quantity must be
  positive")
  val prefix = "http://localhost:8080/random/int"
  val url =
  "$prefix/from/0/to/${chars.lastIndex}/quantity/$length"
  val restTemplate = RestTemplate()
  val passwords = (1..quantity).map {
   val result = restTemplate.getForObject(
    url, GeneratorResult::class.java
   ) as GeneratorResult<Int>
   String(result.values.map { chars[it] }.toCharArray())
  }
  return successResult(passwords)
 }
}
```

Note that the password generator service will need the definition of the `GeneratorResult` class to represent both its own response and that of the number generator. For a simple case like this, we can just copy this definition

to our second project. In more complex scenarios with a multitude of classes to represent the request/response data, it may be worth using some code sharing; we could've, for example, set up a multi-module project which includes both services and a separate module for the shared classes or extracted shared code into a separate project whose output is published into some artifact repository and then used as a dependency in both services.

Since both our services are running as standalone applications. they'll have to listen on different ports. So, before running the password generator service, make sure that its port doesn't conflict with the first one. For this example, we'll set it to `8081` by changing `application.properties` to:

```
server.port=8081
```

Now, we can start the second service and try to make use of its functionality by querying a URL of the form `localhost:8081/password/length/12/quantity/10`. When processing such a URL, our password service will make multiple requests to the random number generator and use its response to compose a list of passwords. Figure 17.8 shows an example result of accessing the password service via a browser:

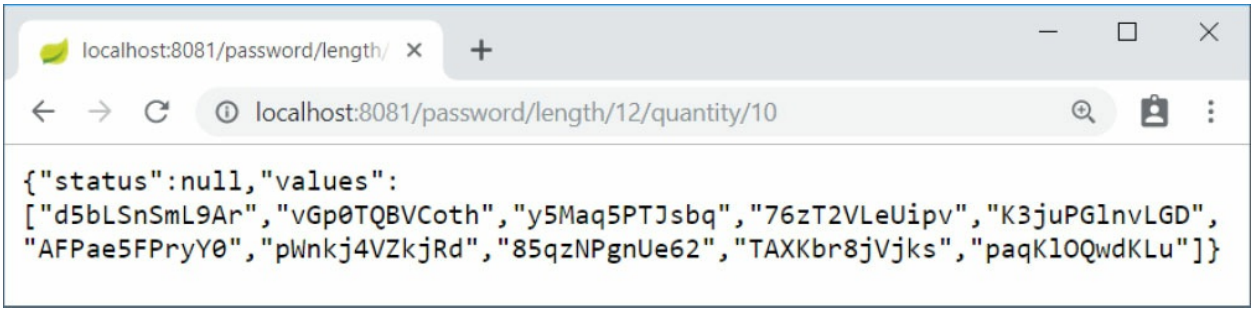

Figure 17.8: *An example of a password generator response*

Note that the password service makes a sequence of requests which are then processed synchronously:

```
val passwords = (1..quantity).map {
 val result = restTemplate.getForObject(
  url,GeneratorResult::class.java
 ) as GeneratorResult<Int>
 String(result.values.map { chars[it] }.toCharArray())
}
```

In other words, the service thread becomes blocked each time you call the

`getForObject()` methods and is unable to do any useful work until it gets all expected responses. This may hinder the service scalability when a number of simultaneous requests grow, so in general we might need to use some asynchronous programming technique such as the Kotlin Coroutines library or reactive frameworks like RxJava or Akka.

With Ktor, as we'll see in the following section, this problem is largely mitigated by the fact that the framework is already built on top of the Coroutines library and provides out-of-the-box support of asynchronous computations via suspending functions. Let's see how our password generator service might look like when implemented using Ktor facilities.

Microservices with Ktor

In the previous chapter, we've introduced you to the Ktor framework which simplifies development of connected client/server applications. In the remaining sections, we're going to extend our knowledge by showing you how Ktor can be used to easily implement a microservice.

The section is composed of two parts. In the first one, we'll introduce one more Ktor feature which deals with JSON-based object serialization both on the client and server sides. This feature allows you to automatically convert Kotlin objects into the corresponding JSON description on sending as well as restore them from JSON on receive similarly to how the Spring framework does that in our earlier example.

In the second part, we'll re-implement the password generator service using the Ktor API. This will allow you to compare Ktor features with their Spring counterparts (e.g. routing DSL vs. request mapping annotations) and also serve as a demonstration of how microservices powered by different frameworks are able to seamlessly communicate with each other.

Using the JSON serialization feature

In the previous chapter, we've seen examples of sending plain text responses using `Ktor'srespondText()` function. Although, we certainly can use it for composing JSON, Ktor provides an easier solution with the `ContentNegotiation` feature which allows you to configure converters for serializing arbitrary objects. In general, to use it for a particular MIME type, you need to register the corresponding implementation of the

`ContentConverter` interface which handles send/receive operations. Ktor comes with an out-of-the-box support of three basic serialization mechanisms:

- Jackson library (**https://github.com/FasterXML/jackson**)
- `google-gson` library (**https://github.com/google/gson**)
- `kotlinx.serialization` (**https://github.com/Kotlin/kotlinx.serialization**)

In our example, we'll use the Jackson-based implementation. Since the corresponding converter belongs to a separate `io.ktor:ktor-jacksonartifact`, make sure you include the necessary dependency in the `build.gradle` file:

```
compile "io.ktor:ktor-jackson:$ktor_version"
```

After that, you can configure the JSON serialization using the `ContentNegotiation` installation block:

```
fun Application.module() {
  …
  install(ContentNegotiation) {
    jackson()
  }
  …
}
```

The preceding `jackson()` function associates `JacksonConverter` with the application/json content type and sets the default behavior of output formatting. Serialization covers both request and response data. For example, we can send some objects into the response and they are automatically converted into the JSON text format:

```
call.respond(successResult(listOf("12345678")))
```

Similarly, we can deserialize JSON objects received with a client's request turning them into ordinary Kotlin objects:

```
data class PasswordSpec(val length: Int, val quantity: Int)
…
val spec = call.receive<PasswordSpec>()
```

In our case, though, we'll be receiving JSON data as a response from another service after making the corresponding request, we also have to configure

serialization for our `HttpClient` instance. Ktor supports the same three serializer implementations for client applications as well. We just need to add respective dependencies to the service `build.gradle.`:

```
compile "io.ktor:ktor-client-json:$ktor_version"
compile "io.ktor:ktor-client-jackson:$ktor_version"
```

To enable serialization on the client-side, we then install the JsonFeature:

```
val client = HttpClient(Apache) {
  …
  install(JsonFeature)
  …
}
```

By default, the particular serializer implementation is chosen automatically based on the included artifact. When necessary, we can also specify it explicitly by assigning an instance of `JsonSerializer` to the serializer property:

```
val client = HttpClient(Apache) {
  …
  install(JsonFeature) {
    serializer = JacksonSerializer()
  }
  …
}
```

Having configured `JsonFeature`, we can automatically read our objects from HTTP responses using the `get()` function:

```
valurl =
"http://localhost:8080/random/int/from/0/to/10/quantity/5"
val result = client.get<GeneratorResult<Int>>(url)
```

Now that we have automatic serialization at our disposal, let's see how we can use it together with other Ktor features for the actual microservice implementation.

Implementing a password generator service

To demonstrate the differences between the Ktor and Spring approach, we'll re-implement the password generator service. Most of the code will expectedly remain the same as both the implementations will follow the same business logic.

To access our first service, we'll use HttpClient instead of RestTemplate:

```
val prefix = "http://localhost:8080/random/int"
val url = "$prefix/from/0/to/${chars.lastIndex}/quantity/$length"
val passwords = (1..quantity).map {
 val result = client.get<GeneratorResult<Int>>(url)
 String(result.values.map { chars[it] }.toCharArray())
}
```

Note that unlike our Spring-based example, this code is asynchronous; the `HttpClient.get()` is a suspending function invoked in a Ktor-supplied coroutine context. As a result, the service threads are not blocked and our server can process further requests while waiting for a response from the random number generator.

The Ktor routing DSL will replace the request dispatching based on the Spring's `@RestController/@RequestMapping` annotations:

```
route("/password") {
 get("/length/{length}/quantity/{quantity}") { … }
}
```

As you can see, the path syntax is basically the same but the use of the DSL allows one to largely eliminate a boilerplate code.

To put our Ktor version of password generator, here is the full text of the server application module:

```
package com.example

import com.fasterxml.jackson.databind.SerializationFeature
import io.ktor.application.*
import io.ktor.client.HttpClient
import io.ktor.client.engine.apache.Apache
import io.ktor.client.features.json.*
import io.ktor.client.request.get
import io.ktor.features.ContentNegotiation
import io.ktor.jackson.jackson
import io.ktor.response.respond
import io.ktor.routing.*

fun main(args: Array<String>): Unit =
 io.ktor.server.netty.EngineMain.main(args)

private val chars = ('a'..'z') + ('A'..'Z') + ('0'..'9')

@Suppress("unused") // Referenced in application.conf
fun Application.module() {
```

```
 install(ContentNegotiation) {
  jackson {
   enable(SerializationFeature.INDENT_OUTPUT)
  }
 }

 val client = HttpClient(Apache) {
  install(JsonFeature) {
   serializer = JacksonSerializer()
  }
 }

 suspend fun ApplicationCall.genPasswords():
 GeneratorResult<String> {
  val length = parameters["length"]?.toIntOrNull()
  ?: return errorResult("Length must be an integer")
  val quantity = parameters["quantity"]?.toIntOrNull()
  ?: return errorResult("Quantity must be an integer")
  if (quantity <= 0) return errorResult("Quantity must be
  positive")
  val prefix = "http://localhost:8080/random/int"
  val url =
  "$prefix/from/0/to/${chars.lastIndex}/quantity/$length"
   val passwords = (1..quantity).map {
   val result = client.get<GeneratorResult<Int>>(url)
   String(result.values.map { chars[it] }.toCharArray())
  }
  return successResult(passwords)
 }
 routing {
  route("/password") {
   get("/length/{length}/quantity/{quantity}") {
    call.respond(call.genPasswords())
   }
  }
 }
}
```

Since our original Spring Boot implementation was listening to the port `8081`, we need to make necessary changes in the Ktor version as well by adjusting its `application.conf` file:

```
ktor {
 deployment {
  port = 8081
  port = ${?PORT}
 }
 application {
```

```
    modules = [ com.example.ApplicationKt.module ]
  }
}
```

Now, if you start both the number and password generator services and open your browser at `localhost:8081/password/length/12/quantity/10`, you'll see a very similar result to the one shown in Figure 17.8 (albeit with a different list of passwords). Note that even though the number generator is based on Spring, the password generator now uses Ktor; both services are easily communicating regardless of their implementational differences.

Conclusion

In this chapter, we got a basic understanding of how you can implement a microservice-based application in Kotlin using either Spring or Ktor frameworks. We explained the key ideas of the microservice architecture, walked you through the setup steps of a simple Spring Boot project and discussed the usage of Spring REST controllers and templates for the purpose of microservice implementation. On top of that, we described how to configure and use JSON serialization in Ktor, a feature which is especially useful for web applications providing some formalized API. Starting from these basics, you can now refine your knowledge by referring to additional resources. We recommend you to begin with guides at the spring.io **(https://spring.io/guides)** as well as Ktor samples we've already mentioned in the previous chapter at **https://ktor.io/samples**.

Questions

1. Explain basic principles of the microservice model.
2. Describe basic steps for setting up a Spring Boot project.
3. What is a Spring controller class? Explain how request data are mapped to controller methods.
4. Compare Spring request mapping with Ktor routing.
5. How do you configure JSON serialization in Ktor?
6. Give an example of microservice implementation using both Spring Boot and Ktor.