Chapter 1. Kotlin Essentials

Kotlin was created by the JetBrains team from St. Petersburg, Russia. JetBrains is perhaps best known for the IntelliJ Idea IDE, the basis for Android Studio. Kotlin is now used in a wide variety of environments across multiple operating systems. It has been nearly five years since Google announced support for Kotlin on Android. According to the Android Developers Blog, as of 2021, over 1.2 million apps in the Google Play store use Kotlin, including 80% of the top one thousand apps.

If you've picked up this book, we are assuming that you are already an Android developer and that you are familiar with Java.

Kotlin was designed to interoperate with Java. Even its name, taken from an island near St. Petersburg, is a sly allusion to Java, an island in Indonesia. Though Kotlin supports other platforms (iOS, WebAssembly, Kotlin/JS, etc.), a key to Kotlin's broad use is its support for the Java virtual machine (JVM). Since Kotlin can be compiled to Java bytecode, it can run anywhere that a JVM runs.

Much of the discussion in this chapter will compare Kotlin to Java. It's important to understand, though, that Kotlin is not just warmed-over Java with some added bells and whistles. Kotlin is a new and different language with connections to Scala, Swift, and C# that are nearly as strong as its connection with Java. It has its own styles and its own idioms. While it is possible to think Java and write Kotlin, thinking in idiomatic Kotlin will reveal the full power of the language.

We realize that there may be some Android developers who have been working with Kotlin for some time, and who have never written any Java at all. If this sounds like you, you may be able to skim this chapter and its review of the Kotlin language. However, even if you are fairly handy with the language, this may be a good chance to remind yourself of some of the details.

This chapter isn't meant to be a full-fledged primer on Kotlin, so if you are completely new to Kotlin, we recommend the excellent *Kotlin in Action*.[1] Instead, this chapter is a review of some Kotlin basics: the type

system, variables, functions, and classes. Even if you are not a Kotlin language expert, it should provide enough of a foundation for you to understand the rest of the book.

As with all statically typed languages, Kotlin's type system is the meta language that Kotlin uses to describe itself. Because it is an essential aspect for discussing Kotlin, we'll start by reviewing it.

The Kotlin Type System

Like Java, Kotlin is a statically typed language. The Kotlin compiler knows the type of every entity that a program manipulates. It can make deductions[2] about those entities and, using those deductions, identify errors that will occur when code contradicts them. Type checking allows a compiler to catch and flag an entire large class of programming errors. This section highlights some of the most interesting features of Kotlin's type system, including the `Unit` type, functional types, null safety, and generics.

Primitive Types

The most obvious difference between Java's and Kotlin's type systems is that Kotlin has no notion of a *primitive type*.

Java has the types `int`, `float`, `boolean`, etc. These types are peculiar in that they do not inherit from Java's base type, `Object`. For instance, the statement `int n = null;` is not legal Java. Neither is `List<int> integers;`. In order to mitigate this inconsistency, each Java primitive type has a *boxed type* equivalent. `Integer`, for instance, is the analog of `int`; `Boolean` of `boolean`; and so on. The distinction between primitive and boxed types has nearly vanished because, since Java 5, the Java compiler automatically converts between the boxed and unboxed types. It is now legal to say `Integer i = 1`.

Kotlin does not have primitive types cluttering up its type system. Its single base type, `Any`, analogous to Java's `Object`, is the root of the entire Kotlin type hierarchy.

NOTE

Kotlin's internal representation of simple types is not connected to its type system. The Kotlin compiler has sufficient information to represent, for instance, a 32-bit integer with as much efficiency as any other language. So, writing `val i: Int = 1` might result in using a primitive type or a boxed type, depending on how the `i` variable is used in the code. Whenever possible, the Kotlin compiler will use primitive types.

Null Safety

A second major difference between Java and Kotlin is that *nullability* is part of Kotlin's type system. A nullable type is distinguished from its nonnullable analog by the question mark at the end of its name; for example, `String` and `String?`, `Person` and `Person?`. The Kotlin compiler will allow the assignment of `null` to a nullable type: `var name: String? = null`. It will not, however, permit `var name: String = null` (because `String` is not a nullable type).

`Any` is the root of the Kotlin type system, just like `Object` in Java. However, there's a significant difference: `Any` is the base class for all nonnullable classes, while `Any?` is the base class for all nullable ones. This is the basis of *null safety*. In other words, it may be useful to think of Kotlin's type system as two identical type trees: all nonnullable types are subtypes of `Any` and all nullable types are subtypes of `Any?`.

Variables must be initialized. There is no default value for a variable. This code, for instance, will generate a compiler error:

```
val name: String // error! Nonnullable types must be initialized!
```

As described earlier, the Kotlin compiler makes deductions using type information. Often the compiler can figure out the type of an identifier from information it already has. This process is called *type inference*. When the compiler can infer a type, there is no need for the developer to specify it. For instance, the assignment `var name = "Jerry"` is perfectly legal, despite the fact that the type of the variable `name` has not been specified. The compiler can infer that the variable `name` must be a `String` because it is assigned the value `"Jerry"` (which is a `String`).

Inferred types can be surprising, though. This code will generate a compiler error:

```
var name = "Jerry"
name = null
```

The compiler inferred the type `String` for the variable `name`, not the type `String?`. Because `String` is not a nullable type, attempting to assign `null` to it is illegal.

It is important to note that a *nullable* type is not the same as its *nonnullable* counterpart. As makes sense, a nullable type behaves as the supertype of the related nonnullable type. This code, for instance, compiles with no problem because a `String` is a `String?`:

```
val name = Jerry
fun showNameLength(name: String?) { // Function accepts a nullable paramet
     // ...
}

showNameLength(name)
```

On the other hand, the following code will not compile at all, because a `String?` is *not* a `String`:

```
val name: String? = null
fun showNameLength(name: String) { // This function only accepts non-nulls
    println(name.length)
}

showNameLength(name)               // error! Won't compile because "name"
                                   // can be null
```

Simply changing the type of the parameter will not entirely fix the problem:

```
val name: String? = null
fun showNameLength(name: String?) { // This function now accepts nulls
    println(name.length)            // error!
}
```

```
showNameLength(name)                    // Compiles
```

This snippet fails with the error `Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?`.

Kotlin requires that nullable variables be handled safely—in a way that cannot generate a null pointer exception. In order to make the code compile, it must correctly handle the case in which `name` is `null`:

```
val name: String? = null
fun showNameLength(name: String?) {
    println(if (name == null) 0 else name.length)
    // we will use an even nicer syntax shortly
}
```

Kotlin has special operators, `?.` and `?:`, that simplify working with nullable entities:

```
val name: String? = null
fun showNameLength(name: String?) {
    println(name?.length ?: 0)
}
```

In the preceding example, when `name` is not `null`, the value of `name?.length` is the same as the value of `name.length`. When `name` is `null`, however, the value of `name?.length` is `null`. The expression does not throw a null pointer exception. Thus, the first operator in the previous example, the safe operator `?.`, is syntactically equivalent to:

```
if (name == null) null else name.length
```

The second operator, the *elvis operator* `?:`, returns the left expression if it is non-null, or the right expression otherwise. Note that the expression on the right-hand side is evaluated only if the left expression is null.

It is equivalent to:

```
if (name?.length == null) 0 else name.length
```

The Unit Type

In Kotlin, *everything* has a value. Always. Once you understand this, it is not difficult to imagine that even a method that doesn't specifically return anything has a default value. That default value is named `Unit`. `Unit` is the name of exactly one object, the value things have if they don't have any other value. The type of the `Unit` object is, conveniently, named `Unit`.

The whole concept of `Unit` can seem odd to Java developers who are used to a distinction between expressions—things that have a value—and statements—things that don't.

Java's conditional is a great example of the distinction between a *statement* and an *expression* because it has one of each! In Java you can say:

```
if (maybe) doThis() else doThat();
```

You cannot, however, say:

```
int n = if (maybe) doThis() else doThat();
```

Statements, like the `if` statement, do not return a value. You cannot assign the value of an `if` statement to a variable, because `if` statements don't return anything. The same is true for loop statements, case statements, and so on.

Java's `if` statement, however, has an analog, the *ternary expression*. Since it is an expression, it returns a value and that value can be assigned. This is legal Java (provided both `doThis` and `doThat` return integers):

```
int n = (maybe) ? doThis() : doThat();
```

In Kotlin, there is no need for two conditionals because `if` is an expression and returns a value. For example, this is perfectly legal:

```
val n = if (maybe) doThis() else doThat()
```

In Java, a method with `void` as the return type is like a statement. Actually, this is a bit of a misnomer because `void` isn't a type. It is a reserved word in the Java language that indicates that the method does not return a value. When Java introduced generics, it introduced the type `Void` to fill the void (intended!). The two representations of "nothing," the keyword and the type, however, are confusing and inconsistent: a function whose return type is `Void` must explicitly return `null`.

Kotlin is much more consistent: all functions return a value and have a type. If the code for a function does not return a value explicitly, the function has the value `Unit`.

Function Types

Kotlin's type system supports *function types*. For example, the following code defines a variable, `func`, whose value is a function, the lambda `{ x -> x.pow(2.0) }`:

```
val func: (Double) -> Double = { x -> x.pow(2.0) }
```

Since `func` is a function that takes one `Double` type argument and returns a `Double`, it's type is `(Double) -> Double`.

In the previous example, we specified the type of `func` explicitly. However, the Kotlin compiler can infer a lot about the type of the variable `func` from the value assigned to it. It knows the return type because it knows the type of `pow`. It doesn't, however, have enough information to guess the type of the parameter `x`. If we supply that, though, we can omit the type specifier for the variable:

```
val func = { x: Double -> x.pow(2.0)}
```

NOTE

Java's type system cannot describe a function type—there is no way to talk about functions outside the context of the classes that contain them. In Java, to do something similar to the previous example, we would use the functional type `Function`, like this:

```
Function<Double, Double> func
    = x -> Math.pow(x, 2.0);

func.apply(256.0);
```

The variable `func` has been assigned an anonymous instance of the type `Function` whose method `apply` is the given lambda.

Thanks to function types, functions can receive other functions as parameters or return them as values. We call these *higher-order functions*. Consider a template for a Kotlin type: `(A, B) -> C`. It describes a function that takes two parameters, one of type `A` and one of type `B` (whatever types those may be), and returns a value of type `C`. Because Kotlin's type language can describe functions, `A`, `B`, and `C` can all, themselves, be functions.

If that sounds rather meta, it's because it is. Let's make it more concrete. For `A` in the template, let's substitute `(Double, Double)` `->` `Int`. That's a function that takes two `Double`s and returns an `Int`. For `B`, let's just substitute a `Double`. So far, we have `((Double, Double)` `->` `Int,` `Double)` `->` `C`.

Finally, let's say our new functional type returns a `(Double)` `->` `Int`, a function that takes one parameter, a `Double`, and returns an `Int`. The following code shows the complete signature for our hypothetical function:

```
fun getCurve(
    surface: (Double, Double) -> Int,
    x: Double
): (Double) -> Int {
    return { y -> surface(x, y) }
}
```

We have just described a function type that takes two arguments. The first is a function (`surface`) of two parameters, both `Double`s, that returns an `Int`. The second is a `Double` (`x`). Our `getCurve` function returns a function that takes one parameter, a `Double` (`y`), and returns an `Int`.

The ability to pass functions as arguments into other functions is a pillar of functional languages. Using higher-order functions, you can reduce code redundancy, while not having to create new classes as you would in Java (subclassing `Runnable` or `Function` interfaces). When used wisely, higher-order functions improve code readability.

Generics

Like Java, Kotlin's type system supports type variables. For instance:

```
fun <T> simplePair(x: T, y: T) = Pair(x, y)
```

This function creates a Kotlin `Pair` object in which both of the elements must be of the same type. Given this definition, `simplePair("Hello", "Goodbye")` and `simplePair(4, 5)` are both legal, but `simplePair("Hello", 5)` is not.

The generic type denoted as `T` in the definition of `simplePair` is a type variable: the values it can take are Kotlin types (in this example, `String` or `Int`). A function (or a class) that uses a type variable is said to be *generic*.

Variables and Functions

Now that we have Kotlin's type language to support us, we can start to discuss the syntax of Kotlin itself.

In Java the top-level syntactic entity is the class. All variables and methods are members of some class or other, and the class is the main element in a homonymous file.

Kotlin has no such limitations. You can put your entire program in one file, if you like (please don't). You can also define variables and functions outside any class.

Variables

There are two ways to declare a variable: with the keywords `val` and `var`. The keyword is required, is the first thing on the line, and introduces the declaration:

```
val ronDeeLay = "the night time"
```

The keyword `val` creates a variable that is read-only: it cannot be reassigned. Be careful, though! You might think `val` is like a Java variable declared using the `final` keyword. Though similar, it is not the same! Although it cannot be reassigned, a `val` definitely can change value! A `val` variable in Kotlin is more like a Java class's field, which has a getter but no setter, as shown in the following code:

```
val surprising: Double
    get() = Math.random()
```

Every time `surprising` is accessed, it will return a different random value. This is an example of a property with no *backing field*. We'll cover properties later in this chapter. On the other hand, if we had written `val rand = Random()`, then `rand` wouldn't change in value and would be more like a `final` variable in Java.

The second keyword, `var`, creates a familiar mutable variable: like a little box that holds the last thing that was put into it.

In the next section, we will move on to one of Kotlin's features as a functional language: *lambdas*.

Lambdas

Kotlin supports function literals: lambdas. In Kotlin, lambdas are always surrounded by curly braces. Within the braces, the argument list is to the

left of an arrow, `->`, and the expression that is the value of executing the lambda is to the right, as shown in the following code:

```
{ x: Int, y: Int -> x * y }
```

By convention, the returned value is the value of the last expression in the body of the lambda. For example, the function shown in the following code is of type `(Int, Int) -> String`:

```
{ x: Int, y: Int -> x * y; "down on the corner" }
```

Kotlin has a very interesting feature that allows actually extending the language. When the last argument to a function is another function (the function is higher-order), you can move the lambda expression passed as a parameter out of the parentheses that normally delimit the actual parameter list, as shown in the following code:

```
// The last argument, "callback", is a function
fun apiCall(param: Int, callback: () -> Unit)
```

This function would typically be used like this:

```
apiCall(1, { println("I'm called back!")})
```

But thanks to the language feature we mentioned, it can also be used like this:

```
apiCall(1) {
   println("I'm called back!")
}
```

This is much nicer, isn't it? Thanks to this feature, your code can be more readable. A more advanced usage of this feature are `DSL`s.[3]

Extension Functions

When you need to add a new method to an existing class, and that class comes from a dependency whose source code you don't own, what do you do?

In Java, if the class isn't `final`, you can subclass it. Sometimes this isn't ideal, because there's one more type to manage, which adds complexity to the project. If the class is `final`, you can define a static method inside some utility class of your own, as shown in the following code:

```
class FileUtils {
    public static String getWordAtIndex(File file, int index) {
        /* Implementation hidden for brevity */
    }
}
```

In the previous example, we defined a function to get a word in a text file, at a given index. On the use site, you'd write `String word = getWordAtIndex(file, 3)`, assuming you make the static import of `FileUtils.getWordAtIndex`. That's fine, we've been doing that for years in Java, and it works.

In Kotlin, there's one more thing you can do. You have the ability to define a new method on a class, even though it isn't a real member-function of that class. So you're not really extending the class, but on the use site it feels like you added a method to the class. How is this possible? By defining an *extension function*, as shown in the following code:

```
// declared inside FileUtils.kt
fun File.getWordAtIndex(index: Int): String {
    val context = this.readText()  // 'this' corresponds to the file
    return context.split(' ').getOrElse(index) { "" }
}
```

From inside the declaration of the extension function, `this` refers to the receiving type instance (here, a `File`). You only have access to public and internal attributes and methods, so `private` and `protected` fields are inaccessible—you'll understand why shortly.

On the use site, you would write `val word = file.getWordAtIndex(3)`. As you can see, we invoke the

`getWordAtIndex()` function on a `File` instance, as if the `File` class had the `getWordAtIndex()` member-function. That makes the use site more expressive and readable. We didn't have to come up with a name for a new utility class: we can declare extension functions directly at the root of a source file.

NOTE

Let's have a look at the decompiled version of `getWordAtIndex`:

```
public class FileUtilsKt {
    public static String getWordAtIndex(
            File file, int index
    ) {
        /* Implementation hidden for brevity */
    }
}
```

When compiled, the generated bytecode of our extension function is the equivalent of a static method which takes a `File` as its first argument. The enclosing class, `FileUtilsKt`, is named after the name of the source file (*FileUtils.kt*) with the "kt" suffix.

That explains why we can't access private fields in an extension function: we are just adding a static method that takes the receiving type as a parameter.

There's more! For class attributes, you can declare *extension properties*. The idea is exactly the same—you're not really extending a class, but you can make new attributes accessible using the dot notation, as shown in the following code:

```
// The Rectangle class has width and height properties
val Rectangle.area: Double
    get() = width * height
```

Notice that this time we used `val` (instead of `fun`) to declare the extension property. You would use it like so: `val area = rectangle.area`.

Extension functions and extension properties allow you to extend classes' capabilities, with a nice dot-notation usage, while still preserving separa-

tion of concern. You're not cluttering existing classes with specific code for particular needs.

Classes

Classes in Kotlin, at first, look a lot like they do in Java: the `class` keyword, followed by the block that defines the class. One of Kotlin's killer features, though, is the syntax for the constructor and the ability to declare properties within it. The following code shows the definition of a simple `Point` class along with a couple of uses:

```
class Point(val x: Int, var y: Int? = 3)

fun demo() {
    val pt1 = Point(4)
    assertEquals(3, pt1.y)
    pt1.y = 7
    val pt2 = Point(7, 7)
    assertEquals(pt2.y, pt1.y)
}
```

Class Initialization

Notice that in the preceding code, the constructor of `Point` is embedded in the declaration of the class. It is called the *primary constructor*. `Point`'s primary constructor declares two class properties, `x` and `y`, both of which are integers. The first, `x`, is read-only. The second, `y`, is mutable and nullable, and has a default value of 3.

Note that the `var` and `val` keywords are very significant! The declaration `class Point(x: Int, y: Int)` is *very* different from the preceding declaration because it does not declare any member properties. Without the keywords, identifiers `x` and `y` are simply arguments to the constructor. For example, the following code will generate an error:

```
class Point(x: Int, y: Int?)

fun demo() {
    val pt = Point(4)
```

```
    pt.y = 7 // error!  Variable expected
}
```

The `Point` class in this example has only one constructor, the one defined in its declaration. Classes are not limited to this single constructor, though. In Kotlin, you can also define both secondary constructors and initialization blocks, as shown in the following definition of the `Segment` class:

```
class Segment(val start: Point, val end: Point) {
    val length: Double = sqrt(
            (end.x - start.x).toDouble().pow(2.0)
                    + (end.y - start.y).toDouble().pow(2.0))

    init {
        println("Point starting at $start with length $length")
    }

    constructor(x1: Int, y1: Int, x2: Int, y2: Int) :
            this(Point(x1, y1), Point(x2, y2)) {
        println("Secondary constructor")
    }
}
```

There are some other things that are of interest in this example. First of all, note that a secondary constructor must delegate to the primary constructor, the `: this(...)`, in its declaration. The constructor may have a block of code, but it is required to delegate, explicitly, to the primary constructor, first.

Perhaps more interesting is the order of execution of the code in the preceding declaration. Suppose one were to create a new `Segment`, using the secondary constructor. In what order would the print statements appear?

Well! Let's try it and see:

```
>>> val s = Segment(1, 2, 3, 4)

Point starting at Point(x=1, y=2) with length 2.82842712
Secondary constructor
```

This is pretty interesting. The `init` block is run before the code block associated with secondary constructor! On the other hand, the properties `length` and `start` have been initialized with their constructor-supplied values. That means that the primary constructor must have been run even before the `init` block.

In fact, Kotlin guarantees this ordering: the primary constructor (if there is one) is run first. After it finishes, `init` blocks are run in declaration order (top to bottom). If the new instance is being created using a secondary constructor, the code block associated with that constructor is the last thing to run.

Properties

Kotlin variables, declared using `val` or `var` in a constructor, or at the top level of a class, actually define a *property*. A property, in Kotlin, is like the combination of a Java field and its getter (if the property is read-only, defined with `val`), or its getter and setter (if defined with `var`).

Kotlin supports customizing the accessor and mutator for a property and has special syntax for doing so, as shown here in the definition of the class `Rectangle`:

```
class Rectangle(val l: Int, val w: Int) {
    val area: Int
        get() = l * w
}
```

The property `area` is *synthetic*: it is computed from the values for the length and width. Because it wouldn't make sense to assign to `area`, it is a `val`, read-only, and does not have a `set()` method.

Use standard "dot" notation to access the value of a property:

```
val rect = Rectangle(3, 4)
assertEquals(12, rect.area)
```

In order to further explore custom property getters and setters, consider a class that has a hash code that is used frequently (perhaps instances are

kept in a `Map`), and that is quite expensive to calculate. As a design decision, you decide to cache the hash code, and to set it when the value of a class property changes. A first try might look something like this:

```
// This code doesn't work (we'll see why)
class ExpensiveToHash(_summary: String) {

    var summary: String = _summary
        set(value) {
            summary = value    // unbounded recursion!!
            hashCode = computeHash()
        }

    //  other declarations here...
    var hashCode: Long = computeHash()

    private fun computeHash(): Long = ...
}
```

The preceding code will fail because of unbounded recursion: the assignment to `summary` is a call to `summary.set()`! Attempting to set the value of the property inside its own setter won't work. Kotlin uses the special identifier `field` to address this problem. The following shows the corrected version of the code:

```
class ExpensiveToHash(_summary: String) {

    var summary: String = _summary
        set(value) {
            field = value
            hashCode = computeHash()
        }

    //  other declarations here...
    var hashCode: Long = computeHash()

    private fun computeHash(): Long = ...
}
```

The identifier `field` has a special meaning only within the custom getter and setter, where it refers to the *backing field* that contains the property's state.

Notice, also, that the preceding code demonstrates the idiom for initializing a property that has a custom getter/setter with a value provided to the class constructor. Defining properties in a constructor parameter list is really handy shorthand. If a few property definitions in a constructor had custom getters and setters, though, it could make the constructor really hard to read.

When a property with a custom getter and setter must be initialized from the constructor, the property is defined, along with its custom getter and setter, in the body of the class. The property is initialized with a parameter from the constructor (in this case, `_summary`). This illustrates, again, the importance of the keywords `val` and `var` in a constructor's parameter list. The parameter `_summary` is just a parameter, not a class property, because it is declared without either keyword.

lateinit Properties

There are times when a variable's value is not available at the site of its declaration. An obvious example of this for Android developers is a UI widget used in an `Activity` or `Fragment`. It is not until the `onCreate` or `onCreateView` method runs that the variable, used throughout the activity to refer to the widget, can be initialized. The `button` in this example, for instance:

```
class MyFragment: Fragment() {
    private var button: Button? = null // will provide actual value later
}
```

The variable must be initialized. A standard technique, since we can't know the value, yet, is to make the variable nullable and initialize it with `null`.

The first question you should ask yourself in this situation is whether it is really necessary to define this variable at this moment and at this location. Will the `button` reference really be used in several methods or is it really only used in one or two specific places? If the latter, you can eliminate the class global altogether.

However, the problem with using a nullable type is that whenever you use `button` in your code, you will have to check for nullability. For example: `button?.setOnClickListener { .. }`. A couple of variables like this and you'll end up with a lot of annoying question marks! This can look particularly cluttered if you are used to Java and its simple dot notation.

Why, you might ask, does Kotlin prevent me from declaring the `button` using a non-null type when you are *sure* that you will initialize it before anything tries to access it? Isn't there a way to relax the compiler's initialization rule just for this `button`?

It's possible. You can do exactly that using the `lateinit` modifier, as shown in the following code:

```
class MyFragment: Fragment() {
    private lateinit var button: Button // will initialize later
}
```

Because the variable is declared `lateinit`, Kotlin will let you declare it without assigning it a value. The variable must be mutable, a `var`, because, by definition, you will assign a value to it, later. Great—problem solved, right?

We, the authors, thought exactly that when we started using Kotlin. Now, we lean toward using `lateinit` only when absolutely necessary, and using nullable values instead. Why?

When you use `lateinit`, you're telling the compiler, "I don't have a value to give you right now. But I'll give you a value later, I promise." If the Kotlin compiler could talk, it would answer, "Fine! You say you know what you're doing. If something goes wrong, it's on you." By using the `lateinit` modifier, you disable Kotlin's null safety for your variable. If you forget to initialize the variable or try to call some method on it before it's initialized, you'll get an `UninitializedPropertyAccessException`, which is essentially the same as getting a `NullPointerException` in Java.

Every single time we've used `lateinit` in our code, we've been burned eventually. Our code might work in all of the cases we'd foreseen. We've

been certain that we didn't miss anything… and we were wrong.

When you declare a variable `lateinit` you're making assumptions that the compiler cannot prove. When you or other developers refactor the code afterward, your careful design might get broken. Tests might catch the error. Or not.[4] In our experience, using `lateinit` always resulted in runtime crashes. How did we fix that? By using a nullable type.

When you use a nullable type instead of `lateinit`, the Kotlin compiler will force you to check for nullability in your code, exactly in the places that it might be null. Adding a few question marks is definitely worth the trade-off for more robust code.

Lazy Properties

It's a common pattern in software engineering to put off creating and initializing an object until it is actually needed. This pattern is known as *lazy initialization*, and is especially common on Android, since allocating a lot of objects during app startup can lead to a longer startup time.
Example 1-1 is a typical case of lazy initialization in Java.

Example 1-1. Java lazy initialization

```
class Lightweight {
    private Heavyweight heavy;

    public Heavyweight getHeavy() {
        if (heavy == null) {
            heavy = new Heavyweight();
        }
        return heavy;
    }
}
```

The field `heavy` is initialized with a new instance of the class `Heavyweight` (which is, presumably, expensive to create) only when its value is first requested with a call, for example, to `lightweight.getHeavy()`. Subsequent calls to `getHeavy()` will return the cached instance.

In Kotlin, lazy initialization is a part of the language. By using the directive `by lazy` and providing an initialization block, the rest of the lazy instantiation is implicit, as shown in Example 1-2.

Example 1-2. Kotlin lazy initialization

```
class Lightweight {
    val heavy by lazy { // Initialization block
        Heavyweight()
    }
}
```

We will explain this syntax in greater detail in the next section.

NOTE

Notice that the code in Example 1-1 isn't thread-safe. Multiple threads calling `Lightweight`'s `getHeavy()` method simultaneously might end up with different instances of `Heavyweight`.

By default, the code in Example 1-2 is thread-safe. Calls to `Lightweight::getHeavy()` will be synchronized so that only one thread at a time is in the initialization block.

Fine-grained control of concurrent access to a lazy initialization block can be managed using `LazyThreadSafetyMode`.

A Kotlin lazy value will not be initialized until a call is made at runtime. The first time the property `heavy` is referenced, the initialization block will be run.

Delegates

Lazy properties are an example of a more general Kotlin feature, called *delegation*. A declaration uses the keyword `by` to define a delegate that is responsible for getting and setting the value of the property. In Java, one could accomplish something similar with, for example, a setter that passed its argument on as a parameter to a call to a method on some other object, the delegate.

Because Kotlin's lazy initialization feature is an excellent example of the power of idiomatic Kotlin, let's take a minute to unpack it.

The first part of the declaration in Example 1-2 reads `val heavy`. This is, we know, the declaration of a read-only variable, `heavy`. Next comes the keyword `by`, which introduces a delegate. The keyword `by` says that the next identifier in the declaration is an expression that will evaluate to the object that will be responsible for the value of `heavy`.

The next thing in the declaration is the identifier `lazy`. Kotlin is expecting, an expression. It turns out that `lazy` is just a function! It is a function that takes a single argument, a lambda, and returns an object. The object that it returns is a `Lazy<T>` where `T` is the type returned by the lambda.

The implementation of a `Lazy<T>` is quite simple: the first time it is called it runs the lambda and caches its value. On subsequent calls it returns the cached value.

Lazy delegation is just one of many varieties of *property delegation*. Using keyword `by`, you can also define *observable properties* (see the Kotlin documentation for delegated properties). Lazy delegation is, though, the most common property delegation used in Android code.

Companion Objects

Perhaps you are wondering what Kotlin did with static variables. Have no fear; Kotlin uses *companion objects*. A companion object is a *singleton object* always related to a Kotlin class. Although it isn't required, most often the definition of a companion object is placed at the bottom of the related class, as shown here:

```
class TimeExtensions {
    //  other code

    companion object {
        const val TAG = "TIME_EXTENSIONS"
    }
}
```

Companion objects can have names, extend classes, and inherit interfaces. In this example, `TimeExtension`'s companion object is named `StdTimeExtension` and inherits the interface `Formatter`:

```
interface Formatter {
    val yearMonthDate: String
}

class TimeExtensions {
    //  other code

    companion object StdTimeExtension : Formatter {
        const val TAG = "TIME_EXTENSIONS"
        override val yearMonthDate = "yyyy-MM-d"
    }
}
```

When referencing a member of a companion object from outside a class that contains it, you must qualify the reference with the name of the containing class:

```
val timeExtensionsTag = TimeExtensions.StdTimeExtension.TAG
```

A companion object is initialized when Kotlin loads the related class.

Data Classes

There is a category of classes so common that, in Java, they have a name: they are called *POJOs*, or plain old Java objects. The idea is that they are simple representations of structured data. They are a collection of data members (fields), most of which have getters and setters, and just a few other methods: `equals`, `hashCode`, and `toString`. These kinds of classes are so common that Kotlin has made them part of the language. They are called *data classes*.

We can improve our definition of the `Point` class by making it a data class:

```
data class Point(var x: Int, var y: Int? = 3)
```

What's the difference between this class, declared using the `data` modifier, and the original, declared without it? Let's try a simple experiment, first using the original definition of `Point` (without the `data` modifier):

```
class Point(var x: Int, var y: Int? = 3)

fun main() {
    val p1 = Point(1)
    val p2 = Point(1)
    println("Points are equals: ${p1 == p2}")
}
```

The output from this small program will be `"Points are equals: false"`. The reason for this perhaps unexpected result is that Kotlin compiles `p1 == p2` as `p1.equals(p2)`. Since our first definition of the `Point` class did not override the `equals` method, this turns into a call to the `equals` method in `Point`'s base class, `Any`. `Any`'s implementation of `equals` returns `true` only when an object is compared to itself.

If we try the same thing with the new definition of `Point` as a data class, the program will print `"Points are equals: true"`. The new definition behaves as intended because a data class automatically includes overrides for the methods `equals`, `hashCode`, and `toString`. Each of these automatically generated methods depends on all of a class's properties.

For example, the `data class` version of `Point` contains an `equals` method that is equivalent to this:

```
override fun equals(o: Any?): Boolean {
    // If it's not a Point, return false
    // Note that null is not a Point
    if (o !is Point) return false

    // If it's a Point, x and y should be the same
    return x == o.x && y == o.y
}
```

In addition to providing default implementations of `equals` and `hashCode`, a `data class` also provides the `copy` method. Here's an

example of its use:

```
data class Point(var x: Int, var y: Int? = 3)
val p = Point(1)           // x = 1, y = 3
val copy = p.copy(y = 2)  // x = 1, y = 2
```

Kotlin's data classes are a perfect convenience for a frequently used idiom.

In the next section, we examine another special kind of class: *enum classes*.

Enum Classes

Remember when developers were being advised that enums were too expensive for Android? Fortunately, no one is even suggesting that anymore: use enum classes to your heart's desire!

Kotlin's enum classes are very similar to Java's enums. They create a class that cannot be subclassed and that has a fixed set of instances. Also as in Java, enums cannot subclass other types but can implement interfaces and can have constructors, properties, and methods. Here are a couple of simple examples:

```
enum class GymActivity {
    BARRE, PILATES, YOGA, FLOOR, SPIN, WEIGHTS
}

enum class LENGTH(val value: Int) {
    TEN(10), TWENTY(20), THIRTY(30), SIXTY(60);
}
```

Enums work very well with Kotlin's `when` expression. For example:

```
fun requiresEquipment(activity: GymActivity) = when (activity) {
    GymActivity.BARRE -> true
    GymActivity.PILATES -> true
    GymActivity.YOGA -> false
    GymActivity.FLOOR -> false
    GymActivity.SPIN -> true
```

```
        GymActivity.WEIGHTS -> true
    }
```

When the `when` expression is used to assign a variable, or as an expression body of a function as in the previous example, it must be *exhaustive*. An exhaustive `when` expression is one that covers every possible value of its argument (in this case, `activity`). A standard way of assuring that a `when` expression is exhaustive is to include an `else` clause. The `else` clause matches any value of the argument that is not explicitly mentioned in its case list.

In the preceding example, to be exhaustive, the `when` expression must accommodate every possible value of the function parameter `activity`. The parameter is of type `GymActivity` and, therefore, must be one of that enum's instances. Because an enum has a known set of instances, Kotlin can determine that all of the possible values are covered as explicitly listed cases and permit the omission of the `else` clause.

Omitting the `else` clause like this has a really nice advantage: if we add a new value to the `GymActivity` enum, our code suddenly won't compile. The Kotlin compiler detects that the `when` expression is no longer exhaustive. Almost certainly, when you add a new case to an enum, you want to be aware of all the places in your code that have to adapt to the new value. An exhaustive `when` expression that does not include an `else` case does exactly that.

NOTE

What happens if a `when` statement need not return a value (for instance, a function in which the `when` statement's value is not the value of the function)?

If the `when` statement is not used as an expression, the Kotlin compiler doesn't force it to be exhaustive. You will, however, get a lint warning (a yellow flag, in Android Studio) that tells you that it is recommended that a `when` expression on enum be exhaustive.

There's a trick that will force Kotlin to interpret any `when` statement as an expression (and, therefore, to be exhaustive). The extension function defined in Example 1-3 forces the `when` statement to return a value, as

we see in Example 1-4. Because it must have a value, Kotlin will insist that it be exhaustive.

Example 1-3. Forcing `when` to be exhaustive

```
val <T> T.exhaustive: T
    get() = this
```

Example 1-4. Checking for an exhaustive `when`

```
when (activity) {
    GymActivity.BARRE -> true
    GymActivity.PILATES -> true
}.exhaustive // error!  when expression is not exhaustive.
```

Enums are a way of creating a class that has a specified, static set of instances. Kotlin provides an interesting generalization of this capability, the *sealed class*.

Sealed Classes

Consider the following code. It defines a single type, `Result`, with exactly two subtypes. `Success` contains a value; `Failure` contains an `Exception`:

```
interface Result
data class Success(val data: List<Int>) : Result
data class Failure(val error: Throwable?) : Result
```

Notice that there is no way to do this with an `enum`. All of the values of an enum must be instances of the same type. Here, though, there are two distinct types that are subtypes of `Result`.

We can create a new instance of either of the two types:

```
fun getResult(): Result = try {
    Success(getDataOrExplode())
} catch (e: Exception) {
```

```
        Failure(e)
    }
```

And, again, a `when` expression is a handy way to manage a `Result`:

```
fun processResult(result: Result): List<Int> = when (result) {
    is Success -> result.data
    is Failure -> listOf()
    else -> throw IllegalArgumentException("unknown result type")
}
```

We've had to add an `else` branch again, because the Kotlin compiler doesn't know that `Success` and `Failure` are the only `Result` subclasses. Somewhere in your program, you might create another subclass of result `Result` and add another possible case. Hence the `else` branch is required by the compiler.

Sealed classes do for types what enums do for instances. They allow you to announce to the compiler that there is a fixed, known set of subtypes (`Success` and `Failure` in this case) for a certain base type (`Result`, here). To make this declaration, use the keyword `sealed` in the declaration, as shown in the following code:

```
sealed class Result
data class Success(val data: List<Int>) : Result()
data class Failure(val error: Throwable?) : Result()
```

Because `Result` is *sealed*, the Kotlin compiler knows that `Success` and `Failure` are the only possible subclasses. Once again, we can remove the `else` from a `when` expression:

```
fun processResult(result: Result): List<Int> = when (result) {
    is Success -> result.data
    is Failure -> listOf()
}
```

Visibility Modifiers

In both Java and Kotlin, visibility modifiers determine the scope of a variable, class, or method. In Java, there are three visibility modifiers:

`private`

References are only visible to the class that they are defined within, and from the outer class if defined in an inner class.

`protected`

References are visible to the class that they are defined within, or any subclasses of that class. In addition, they are also visible from classes in the same package.

`public`

References are visible anywhere.

Kotlin also has these three visibility modifiers. However, there are some subtle differences. While you can only use them with class-member declarations in Java, you can use them with class-member *and* top-level declarations in Kotlin:

`private`

The declaration's visibility depends on where it is defined:

- A class member declared as `private` is visible only in the *class* in which it is defined.
- A top-level `private` declaration is visible only in the *file* in which it is defined.

`protected`

Protected declarations are visible only in the class in which they are defined, and the subclasses thereof.

`public`

References are visible anywhere, just like in Java.

In addition to these three different visibilities, Java has a fourth, *package-private*, making references only visible from classes that are within the

same package. A declaration is package-private when it has no visibility modifiers. In other words, this is the default visibility in Java.

Kotlin has no such concept.[5] This might be surprising, because Java developers often rely on package-private visibility to hide implementation details from other packages within the same module. In Kotlin, packages aren't used for visibility scoping at all—they're just namespaces. Therefore, the default visibility is different in Kotlin—it's *public*.

The fact that Kotlin doesn't have package-private visibility has quite a significant impact on how we design and structure our code. To guarantee a complete encapsulation of declarations (classes, methods, top-level fields, etc.), you can have all these declarations as `private` within the same file.

Sometimes it's acceptable to have several closely related classes split into different files. However, those classes won't be able to access siblings from the same package unless they are `public` or `internal`. What's `internal`? It's the fourth visibility modifier supported by Kotlin, which makes the reference visible anywhere within the containing *module*.[6] From a module standpoint, `internal` is identical to `public`. However, `internal` is interesting when this module is intended as a library—for example, it's a dependency for other modules. Indeed, `internal` declarations aren't visible from modules that import your library. Therefore, `internal` is useful to hide declarations from the outside world.

NOTE

The `internal` modifier isn't meant for visibility scoping inside the module, which is what package-private does in Java. This isn't possible in Kotlin. It is possible to restrict visibility a little more heavy-handedly using the `private` modifier.

Summary

Table 1-1 highlights some of the key differences between Java and Kotlin.

Table 1-1. Differences between Java and Kotlin features

Feature	Java	Kotlin
File contents	A single file contains a single top-level class.	A single file can hold any number of classes, variables, or functions.
Variables	Use `final` to make a variable immutable; variables are mutable by default. Defined at the class level.	Use `val` to make a variable read-only, or `var` for read/write values. Defined at the class level, or may exist independently outside of a class.
Type inferencing	Data types are required. `Date date = new Date();`	Data types can be inferred, like `val date = Date()`, or explicitly defined, like `val date: Date = Date()`.
Boxing and unboxing types	In Java, data primitives like `int` are recommended for more expensive operations, since they are less expensive than boxed types like `Integer`. However, boxed types have lots of useful methods in Java's wrapper classes.	Kotlin doesn't have primitive types out of the box. Everything is an object. When compiled for the JVM, the generated bytecode performs automatic unboxing, when possible.

Feature	Java	Kotlin
Access modifiers	Public and protected classes, functions, and variables can be extended and overridden.	As a functional language, Kotlin encourages immutability whenever possible. Classes and functions are final by default.
Access modifiers in multi-module projects	Default access is package-private.	There is no package-private, and default access is public. New `internal` access provides visibility in the same module.
Functions	All functions are methods.	Kotlin has function types. Function data types look like, for example, `(param: String) -> Boolean`.
Nullability	Any non-primitive object can be null.	Only explicitly nullable references, declared with the `?` suffix on the type, can be set to null: `val date: Date? = new Date()`.
Statics versus constants	The `static` keyword attaches a variable to a class definition, rather than an instance.	There is no `static` keyword. Use a private `const` or a `companion` object.

Congratulations, you just finished a one-chapter covering Kotlin's essentials. Before we start talking about applying Kotlin to Android, we need to discuss Kotlin's built-in library: collections and data transformations.

Understanding the underlying functions of data transformations in Kotlin will give you the necessary foundation needed to understand Kotlin as a functional language.

1 Dmitry Jemerov and Svetlana Isakova. *Kotlin in Action*. Manning, 2017.

2 Kotlin officially calls this type inferencing, which uses a partial phase of the compiler (the frontend component) to do type checking of the written code while you write in the IDE. It's a plug-in for IntelliJ! Fun fact: the entirety of IntelliJ and Kotlin is made of compiler plug-ins.

3 DSL stands for *domain-specific language*. An example of a DSL built in Kotlin is the Kotlin Gradle DSL.

4 You can check whether the `latenit` `button` property is initialized using `this::button.isInitialized`. Relying on developers to add this check in all the right places doesn't solve the underlying issue.

5 At least, as of Kotlin 1.5.20. As we write these lines, Jetbrains is considering adding a package-private visibility modifier to the language.

6 A module is a set of Kotlin files compiled together.