Chapter 1: Getting Started with Kotlin

The bulk of this chapter will be dedicated to basic Kotlin syntax. It is important to be comfortable with a language before we start implementing any design patterns in it.

We'll also briefly discuss what problems design patterns solve and why you should use them in Kotlin. This will be helpful to those who are less familiar with the concept of design patterns. But even for experienced engineers, it may provide an interesting perspective.

This chapter doesn't aim to cover the entire language vocabulary but to get you familiar with some basic concepts and idioms. The following chapters will expose you to even more language features as they become relevant to the design patterns that we'll discuss.

In this chapter, we will cover the following main topics:

- Basic language syntax and features
- Understanding Kotlin code structure
- Type system and `null` safety
- Reviewing Kotlin data structures
- Control flow
- Working with text and loops
- Classes and inheritance
- Extension functions
- Introduction to design patterns

By the end of this chapter, you'll have a knowledge of Kotlin's basics, which will be the foundation for the following chapters.

Technical requirements

To follow the instructions in this chapter, you'll need the following:

- IntelliJ IDEA Community Edition (https://www.jetbrains.com/idea/download/)
- OpenJDK 11 or higher (https://openjdk.java.net/install/)

The code files for this chapter are available at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter01.

Basic language syntax and features

Whether you come from **Java**, **C#**, **Scala**, or any other statically typed programming language, you'll find Kotlin syntax quite familiar. This is not by coincidence but to make the transition to this new language as smooth as possible for those with previous experience in other languages. Besides that familiarity, Kotlin brings a vast amount of features, such as better type safety. As we move ahead, you'll notice that all of them are attempting to solve real-world problems. That pragmatic approach is remarkably consistent across the language. For example, one of the strongest benefits of Kotlin is complete Java interoperability. You can have Java and Kotlin classes alongside each other and freely use any library that is available in Java for a Kotlin project.

To summarize, the goals of the language are as follows:

- **Pragmatic**: Makes things we do often easy to achieve
- **Readable**: Keeps a balance between conciseness and clarity on what the code does
- **Easy to reuse**: Supports adapting code to different situations
- **Safe**: Makes it hard to write code that crashes
- **Interoperable**: Allows the use of existing libraries and frameworks

This chapter will discuss how these goals are achieved.

Multi-paradigm language

Some of the major paradigms in programming languages are procedural, object-oriented, and functional paradigms.

Being pragmatic, Kotlin allows for any of these paradigms. It has classes and inheritance, coming from the object-oriented approach. It has higher-order functions from functional programming. You don't have to wrap everything in classes if you don't want to, though. Kotlin allows you to structure your entire code as just a set of procedures and structs if you need to. You will see how all these approaches come together, as different examples will combine different paradigms to solve the problems discussed.

Instead of covering all aspects of a topic from start to finish, we will be building the knowledge as we go.

Understanding Kotlin code structure

The first thing you'll need to do when you start programming in Kotlin is to create a new file. Kotlin's file extension is usually `.kt`.

Unlike Java, there's no strong relationship between the filename and class name. You can put as many public classes in your file as you want, as long as the classes are related to one another and your file doesn't grow too long to read.

Naming conventions

As a convention, if your file contains a single class, name your file the same as your class.

If your file contains more than one class, then the filename should describe the common purpose of those classes. Use Camel case when naming your files, as per the Kotlin coding conventions: https://kotlinlang.org/docs/coding-conventions.html.

The main file in your Kotlin project should usually be named **`Main.kt`**.

Packages

A **package** is a collection of files and classes that all share a similar purpose or domain. Packages are a convenient way to have all your classes and functions under the same namespace, and often in the same folder. That's the reason Kotlin, similar to many other languages, uses the notion of a package.

The package that the file belongs to is declared using a **`package`** keyword:

```
  package me.soshin
```

Similar to placing classes in files, you can put any package in any directory or file, but if you're mixing Java and Kotlin, Kotlin files should follow Java package rules, as given at https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html.

In purely Kotlin projects, common package prefixes can be omitted from the folder structure. For example, if all your projects are under the **`me.soshin`** package, and part of your application deals with mortgages, you can place your files directly in the **`/mortgages`** folder and not in the **`/me/soshin/mortgages`** folder like Java requires.

There is no need to declare a package for your **`Main.kt`** file.

Comments

Going forward, we will be documenting parts of the code using **Kotlin comments**. Similar to many other programming languages, Kotlin uses **`//`**

for a single-line comment and `/* */` for multiline comments.

Comments are a useful way to provide more context both to other developers and to your future self. Now, let's write our first Kotlin program and discuss how Kotlin's guiding principles are applied to it.

Hello Kotlin

There's no book dedicated to a programming language that can avoid the ubiquitous *Hello World* example. We're certainly not going to challenge that honored tradition.

To begin learning how Kotlin works, let's put the following code in our **`Main.kt`** file and run it:

```
fun main() {
    println("Hello Kotlin")
}
```

When your run this example, for example by pressing the **Run** button in your IntelliJ IDEA, it simply outputs the following:

```
> Hello Kotlin
```

There are some interesting attributes in that piece of code in comparison to the following Java code that does exactly the same:

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

Let's focus on those attributes in the next sections.

No wrapping class

In Java, C#, Scala, and many other languages, it's necessary to wrap every function in a class for it to become executable.

Kotlin, though, has the concept of **package-level functions**. If your function doesn't need to access properties of a class, you don't need to wrap it in a class. It's as simple as that.

We'll discuss package-level functions in more detail in the following chapters.

IMPORTANT NOTE:

From here on, we'll use ellipsis notation (three dots) to indicate that some parts of the code were omitted to focus on the important bits. You can always find the full code examples at the GitHub link for this chapter.

No arguments

Arguments, supplied as an array of strings, are a way to configure your command-line application. In Java, you cannot have a runnable **`main()`** function that doesn't take this array of arguments:

```
  public static void main(String[] args) { ... }
```

But in Kotlin, those are entirely optional.

No static modifier

Some languages use the **`static`** keyword to indicate that a function in a class can be executed without the need to instantiate the class. The **`main()`** function is one such example.

In Kotlin, there's no such limitation. If your function doesn't have any state, you can place it outside of a class, and there is no **`static`** keyword in Kotlin.

A less verbose print function

Instead of the verbose **`System.out.println`** method that outputs a string to the standard output, Kotlin provides us with an alias called **`println()`** that does exactly the same.

No semicolons

In Java, and many other languages, every statement or expression must be terminated with a semicolon, as shown in the following example:

```
System.out.println("Semicolon =>");
```

Kotlin is a pragmatic language. So, instead, it infers during compilation where it should put the semicolons:

```
println("No semicolons! =>")
```

Most of the time, you won't need to put semicolons in your code. They're considered optional.

This is an excellent example of how pragmatic and concise Kotlin is. It sheds lots of fluff and lets you focus on what's important.

IMPORTANT NOTE:

You don't have to write your code in a file for simple snippets. You can also play with the language online: try https://play.kotlinlang.org/ or use a REPL and an interactive shell after installing Kotlin and running **`kotlinc`**.

Understanding types

Previously, we said that Kotlin is a type-safe language. Let's examine the Kotlin type system and compare it to what Java provides.

IMPORTANT NOTE:

The Java examples are for familiarity and not to prove that Kotlin is superior to Java in any way.

Basic types

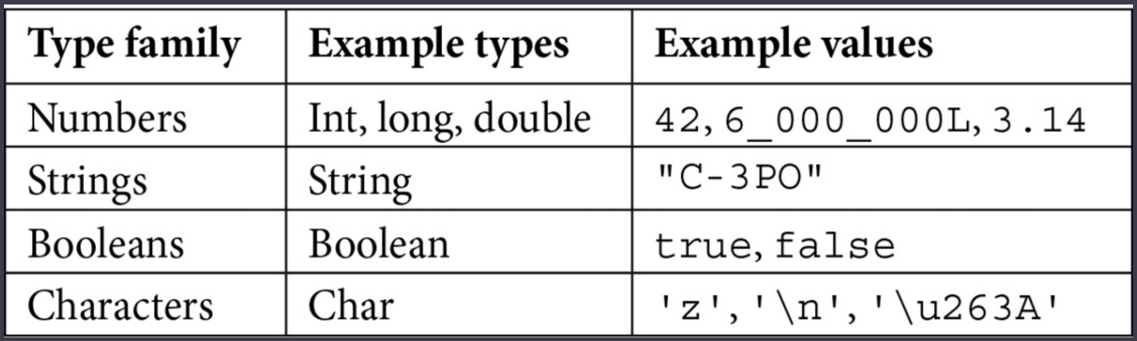
Some languages make a distinction between primitive types and objects. Taking Java as an example, there is the **`int`** type and **`Integer`** – the former being more memory-efficient and the latter more expressive by supporting a lack of value and having methods.

There is no such distinction in Kotlin. From a developer's perspective, all the types are the same.

But it doesn't mean that Kotlin is less efficient than Java in that aspect. The Kotlin compiler optimizes types. So, you don't need to worry about it much.

Most of the Kotlin types are named similarly to Java, the exceptions being Java's **`Integer`** being called **`Int`** and Java's void being called **`Unit`**.

It doesn't make much sense to list all the types, but here are some examples:



Type family	Example types	Example values
Numbers	Int, long, double	42, 6_000_000L, 3.14
Strings	String	"C-3PO"
Booleans	Boolean	true, false
Characters	Char	'z', '\n', '\u263A'

Table 1.1 - Kotlin types

Type inference

Let's declare our first Kotlin variable by extracting the string from our **`Hello Kotlin`** example:

```
var greeting = "Hello Kotlin"
println(greeting)
```

Note that nowhere in our code is it stated that **`greeting`** is of the **`String`** type. Instead, the compiler decides what type of variable should be used.

Unlike interpreted languages, such as JavaScript, Python, or Ruby, the type of variable is defined only once.

In Kotlin, this will produce an error:

```
var greeting = "Hello Kotlin"
greeting = 1 // <- Greeting is a String
```

If you'd like to define the type of variable explicitly, you may use the following notation:

```
var greeting: String = "Hello Kotlin"
```

Values

In Java, variables can be declared `final`. Final variables can be assigned only once and their reference is effectively immutable:

```
final String s = "Hi";
s = "Bye"; // Doesn't work
```

Kotlin urges us to use immutable data as much as possible. Immutable variables in Kotlin are called **values** and use the `val` keyword:

```
val greeting = "Hi"
greeting = "Bye"// Doesn't work, "Val cannot be
reassigned"
```

Values are preferable over variables. Immutable data is easier to reason about, especially when writing concurrent code. We'll touch more on that in *Chapter 5*, *Introducing Functional Programming*.

Comparison and equality

We were taught very early in Java that comparing objects using `==` won't produce the expected results, since it tests for reference equality – whether two pointers are the same, and not whether two objects are equal.

Instead, in Java, we use **`equals()`** for objects and **`==`** to compare only primitives, which may cause some confusion.

JVM does integer caching and string interning to prevent that in some basic cases, so for the sake of the example, we'll use a large integer:

```
Integer a = 1000;
Integer b = 1000;
System.out.println(a == b);      // false
System.out.println(a.equals(b)); // true
```

This behavior is far from intuitive. Instead, Kotlin translates **`==`** to **`equals()`**:

```
val a = 1000
val b = 1000
println(a == b)      // true
println(a.equals(b)) // true
```

If you do want to check for reference equality, use **`===`**. This won't work for some of the basic types, though:

```
println(a === b) // Still true
```

We'll discuss referential equality more when we learn how to instantiate classes.

Declaring functions

In Java, every method must be wrapped by a class or interface, even if it doesn't rely on any information from it. You're probably familiar with many **`Util`** classes in Java that only have static methods, and their only purpose is to satisfy the language requirements and bundle those methods together.

We already mentioned earlier that in Kotlin, a function can be declared outside of a class. We've seen it with the **`main()`** function. The keyword to declare a function is **`fun`**. The argument type comes after the argument name, and not before:

```
fun greet(greeting: String) {
    println(greeting)
}
```

If you need to return a result, its type will come after the function declaration:

```
fun getGreeting(): String {
    return "Hello, Kotlin!"
}
```

You can try this out yourself:

```
fun main() {
    greet(getGreeting())
}
```

If the function doesn't return anything, the return type can be omitted completely. There's no need to declare it as **`void`**, or its Kotlin counterpart, **`Unit`**.

When a function is very short and consists of just a single expression, such as our **`getGreeting()`** function, we can remove the return type and the curly brackets, and use a shorter notation:

```
fun getGreeting() = "Hello, Kotlin!"
```

Here, the Kotlin compiler will infer that we're returning a **`String`** type.

Unlike some scripting languages, the order in which functions are declared is not important. Your **`main`** function will have access to all the other functions in its scope, even if those are declared after it in the code file.

There are many other topics regarding function declarations, such as named arguments, default parameters, and variable numbers of arguments. We'll introduce them in the following chapters with relevant examples.

IMPORTANT NOTE:

Many examples in this book assume that the code we provide is wrapped in the **`main`** *function. If you don't see a signature of the function, it probably should be part of the* **`main`** *function. As an alternative, you can also run the examples in an IntelliJ scratch file.*

Null safety

Probably the most notorious exception in the Java world is **`NullPointerException`**. The reason behind this exception is that every object in Java can be **`null`**. The code here shows us why this is a problem:

```
final String s = null;
System.out.println(s.length());
// Causes NullPointerException
```

It's not like Java didn't attempt to solve that problem, though. Since **Java 8**, there has been an **`Optional`** construct that represents a value that may not be there:

```
var optional = Optional.of("I'm not null");
if (optional.isPresent()) {
    System.out.println(optional.get().length());
}
```

But it doesn't solve our problem. If our function receives **`Optional`** as an argument, we can still pass it a **`null`** value and crash the program at runtime:

```
void printLength(Optional<String> optional) {
    if (optional.isPresent()) { // <- Missing null
check
      here
```

```
          System.out.println(optional.get().length());
      }
  }
  printLength (null); // Crashes!
```

Kotlin checks for nulls during compile time:

```
val s: String = null // Won't compile
```

Let's take a look at the **`printLength()`** function written in Kotlin:

```
fun printLength(s: String) {
    println(s.length)
}
```

Calling this function with **`null`** won't compile at all:

```
printLength(null)
// Null cannot be a value of a non-null type String
```

If you specifically want your type to be able to receive nulls, you'll need to mark it as nullable using the question mark:

```
fun printLength(stringOrNull: String?) { ... }
```

There are multiple techniques in Kotlin for dealing with nulls, such as smart casts, the Elvis operator, and so on. We'll discuss alternatives to nulls in *Chapter 4*, *Getting Familiar with Behavioral Patterns*. Let's now move on to data structures in Kotlin.

Reviewing Kotlin data structures

There are three important groups of data structures we should get familiar with in Kotlin: lists, sets, and maps. We'll cover each briefly, then discuss some other topics related to data structures, such as mutability and tuples.

Lists

A **list** represents an ordered collection of elements of the same type. To declare a list in Kotlin, we use the **`listOf()`** function:

```
val hobbits = listOf("Frodo", "Sam", "Pippin",
"Merry")
```

Note that we didn't specify the type of the list. The reason is that the type inference can also be used when constructing collections in Kotlin, the same as when initializing variables.

If you want to provide the type of the list, you similarly do that for defining arguments for a function:

```
val hobbits: List<String> = listOf("Frodo", "Sam",
"Pippin",   "Merry")
```

To access an element in the list at a particular index, we use square brackets:

```
println(hobbits[1])
```

The preceding code will output this:

```
> Sam
```

Sets

A **set** represents a collection of unique elements. Looking for the presence of an element in a set is much faster than looking it up in a list. But, unlike lists, sets don't provide indexes access.

Let's create a set of football World Cup champions until after 1994:

```
val footballChampions = setOf("France", "Germany",
"Spain",   "Italy", "Brazil", "France", "Brazil",
"Germany")
println(footballChampions) // [France, Germany,
Spain,   Italy, Brazil]
```

You can see that each country exists in a set exactly once. To check whether an element is in a **Set** collection, you can use the **in** function:

```
println("Israel" in footballChampions)
println("Italy" in footballChampions)
```

This gives us the following:

```
> false
> true
```

Note that although sets, in general, do not guarantee the order of elements, the current implementation of a **setOf()** function returns **LinkedHashSet**, which preserves insertion order – **France** appears first in the output, since it was the first country in the input.

Maps

A **map** is a collection of key-value pairs, in which keys are unique. The keyword that creates a pair of two elements is **to**. In fact, this is not a real keyword but a special function. We'll learn about it more in *Chapter 5*, *Introducing Functional Programming*.

In the meantime, let's create a map of some of the Batman movies and the actors that played Bruce Wayne in them:

```
val movieBatmans = mapOf(
    "Batman Returns" to "Michael Keaton",
    "Batman Forever" to "Val Kilmer",
    "Batman & Robin" to "George Clooney"
)
println(movieBatmans)
```

This prints the following:

```
> {Batman Returns=Michael Keaton,
> Batman Forever=Val Kilmer,
> Batman & Robin=George Clooney}
```

To access a value by its key, we use square brackets and provide the key:

```
println(movieBatmans["Batman Returns"])
```

The preceding code will output this:

```
> Michael Keaton
```

Those data structures also support checking that an element doesn't exist:

```
println(" Batman Begins " !in movieBatmans)
```

We get the following output:

```
> true
```

Mutability

All of the data structures we have discussed so far are immutable or, more correctly, read-only.

There are no methods to add new elements to a list we create with the **`listOf()`** function, and we also cannot replace any element:

```
hobbits[0] = "Bilbo " // Unresolved reference!
```

Immutable data structures are great for writing concurrent code. But, sometimes, we still need a collection we can modify. In order to do that, we can use the mutable counterparts of the collection functions:

```
val editableHobbits = mutableListOf("Frodo", "Sam",
  "Pippin", "Merry")
editableHobbits.add("Bilbo")
```

Editable collection types have functions such as **`add()`** that allow us to modify or, in other words, mutate them.

Alternative implementations for collections

If you have worked with JVM before, you may know that there are other implementations of sets and maps. For example, **`TreeMap`** stores the keys in a sorted order.

Here's how you can instantiate them in Kotlin:

```
import java.util.*
// Mutable map that is sorted by its keys
val treeMap = java.util.TreeMap(
    mapOf(
        "Practical Pig" to "bricks",
        "Fifer" to "straw",
        "Fiddler" to "sticks"
    )
)
println(treeMap.keys)
```

We will get the following output:

```
> [Fiddler, Fifer, Practical Pig]
```

Note that the names of the *Three Little Pigs* are ordered alphabetically.

Arrays

There is one other data structure we should cover in this section – **arrays**. In Java, arrays have a special syntax that uses square brackets. For example, an array of strings is declared **`String[]`**, while a list of strings is declared as **`List<String>`**. An element in a Java array is accessed using square brackets, while an element in a list is accessed using the **`get()`** method.

To get the number of elements in an array in Java, we use the **`length()`** method, and to do the same with a collection, we use the **`size()`** method. This is part of Java's legacy and its attempts to resemble C++.

In Kotlin, array syntax is consistent with other types of collections. An array of strings is declared as **`Array<String>`**:

```
val musketeers: Array<String> = arrayOf("Athos",
"Porthos",   "Aramis")
```

This is the first time we see angle brackets in Kotlin code. Similar to Java or TypeScript, the type between them is called **type argument**. It indicates that this array contains strings. We'll discuss this topic in detail in Chapter 4, *Getting Familiar with Behavioral Patterns*, while covering generics.

If you already have a collection and would like to convert it into an array, use the **`toTypedArray`** function:

```
listOf(1, 2, 3, 5).toTypedArray()
```

In terms of its abilities, a Kotlin array is very similar to a list. For example, to get the number of elements in a Kotlin array, we use the same **`size`** property as other collections.

When would you need to use arrays then? One example is accepting arguments in the **`main`** function. Previously, we've seen only main functions without arguments, but sometimes you want to pass them from a command line.

Here's an example of a **`main`** function that accepts arguments from a command line and prints all of them, separated by commas:

```
fun main(args: Array<String>) {
    println(args.joinToString(", "))
}
```

Other cases include invoking Java functions that expect arrays or using **`varargs`** syntax, which we will discuss in Chapter 3, *Understanding Structural Patterns*.

As we are now familiar with some basic data structures, it's time to discuss how we can apply logic to them using **`if`** and **`when`** expressions.

Control flow

You could say that the control flow is the bread and butter of writing programs. We'll start with two conditional expressions, **`if`** and **`when`**.

The if expression

In Java, **`if`** is a statement. Statements do not return any value. Let's look at the following function, which returns one of two possible values:

```
public String getUnixSocketPolling(boolean isBsd) {
    if (isBsd) {
        return "kqueue";
    }
    else {
        return "epoll";
    }
}
```

While this example is easy to follow, in general, having multiple **`return`** statements is considered bad practice because they often make the code harder to comprehend.

We could rewrite this method using Java's **`var`** keyword:

```
public String getUnixSocketPolling(boolean isBsd) {
    var pollingType = "epoll";
    if (isBsd) {
        pollingType = "kqueue";
    }
    return pollingType;
}
```

Now, we have a single **`return`** statement, but we had to introduce a mutable variable. Again, with such a simple example, this is not an issue. But, in general, you should try to avoid mutable shared state as much as possible, since such code is not thread-safe.

Why are we having problems writing that in the first place, though?

Contrary to Java, in Kotlin, **`if`** is an expression, meaning it returns a value. We could rewrite the previous function in Kotlin as follows:

```
fun getUnixSocketPolling(isBsd: Boolean): String {
    return if (isBsd) {
        "kqueue"
    } else {
        "epoll"
    }
}
```

Or we could use a shorter form:

```
fun getUnixSocketPolling(isBsd: Boolean): String
    = if (isBsd) "kqueue" else "epoll"
```

Due to the fact that **`if`** is an expression, we didn't need to introduce any local variables.

Here, we're again making use of single-expression functions and type inference. The important part is that **`if`** returns a value of the **`String`** type. There's no need for multiple return statements or mutable variables whatsoever.

IMPORTANT NOTE:

Single-line functions in Kotlin are very cool and pragmatic, but you should make sure that somebody else other than you understands what they do. Use with care.

The when expression

What if (no pun intended) we want to have more conditions in our **`if`** *statement?*

In Java, we use the **`switch`** statement. In Kotlin, there's a **`when`** expression, which is a lot more powerful, since it can embed some other Kotlin features. Let's create a method that's given a superhero and tells us who their archenemy is:

```
fun archenemy(heroName: String) = when (heroName) {
    "Batman" -> "Joker"
    "Superman" -> "Lex Luthor"
    "Spider-Man" -> "Green Goblin"
    else -> "Sorry, no idea"
}
```

The **`when`** expression is very powerful. In the next chapters, we will elaborate on how we can combine it with ranges, **`enums`**, and **`sealed`** classes as well.

As a general rule, use **`when`** if you have more than two conditions. Use **`if`** for simple cases.

Working with text

We've already seen many examples of working with text in the previous section. After all, it's not possible to print **`Hello Kotlin`** without using a string, or at least it would be very awkward and inconvenient.

In this section, we'll discuss some of the more advanced features that allow you to manipulate text efficiently.

String interpolation

Let's assume now we would like to actually print the results from the previous section.

First, as you may have already noticed, in one of the previous examples, Kotlin provides a nifty **`println()`** standard function that wraps the bulkier **`System.out.println`** command from Java.

But, more importantly, as in many other modern languages, Kotlin supports string interpolation using the **`${}`** syntax. Let's take the example from before:

```
val hero = "Batman"
println("Archenemy of $hero is ${archenemy(hero)}")
```

The preceding code would print as follows:

```
> Archenemy of Batman is Joker
```

Note that if you're interpolating a value of a function, you need to wrap it in curly braces. If it's a variable, curly braces could be omitted.

Multiline strings

Kotlin supports multiline strings, also known as **raw strings**. This feature exists in many modern languages, and was brought to **Java 15** as **text blocks**.

The idea is quite simple. If we want to print a piece of text that spans multiple lines, let's say something from *Alice's Adventures in Wonderland* by Lewis Carroll, one way is to concatenate it:

```
println("Twinkle, Twinkle Little Bat\n" +
    "How I wonder what you're at!\n" +
    "Up above the world you fly,\n" +
    "Like a tea tray in the sky.\n" +
    "Twinkle, twinkle, little bat!\n" +
    "How I wonder what you're at!")
```

While this approach certainly works, it's quite cumbersome.

Instead, we could define the same string literal using triple quotes:

```
println("""Twinkle, Twinkle Little Bat
           How I wonder what you're at!
           Up above the world you fly,
           Like a tea tray in the sky.
           Twinkle, twinkle, little bat!
           How I wonder what you're at!""")
```

This is a much cleaner way to achieve the same goal. If you execute this example, you may be surprised that the poem is not indented correctly. The reason is that multiline strings preserve whitespace characters, such as tabs.

To print the results correctly, we need to add a **`trimIndent()`** invocation:

```
println("""
    Twinkle, Twinkle Little Bat
    How I wonder what you're at!
    """.trimIndent())
```

Multiline strings also have another benefit – there's no need to escape quotes in them. Let's look at the following example:

```
println("From \" Alice's Adventures in Wonderland\"
")
```

Notice how the quote characters that are part of the text had to be escaped using the backslash character.

Now, let's look at the same text using multiline syntax:

```
println(""" From " Alice's Adventures in Wonderland"
""")
```

Note that there's no need for escape characters anymore.

Loops

Now, let's discuss another typical control structure – a **loop**. Loops are a very natural construct for most developers. Without loops, it would be tough to repeat the same code block more than once (although we will discuss how to do that without loops in later chapters).

for-each loop

Probably the most helpful type of a loop in Kotlin is a **for-each** loop. This loop can iterate over strings, data structures, and basically everything that has an iterator. We'll learn more about iterators in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, so for now, let's demonstrate their use on a simple string:

```
for (c in "Word") {
    println(c)
}
```

This will print the following:

```
>W
>o
>r
>d
```

The **for-each** loop works on all the types of data structures we already discussed as well, that is, lists, sets, and maps. Let's take a list as an example:

```
val jokers = listOf("Heath Ledger", "Joaquin
Phoenix",   "Jack Nicholson")
for (j in jokers) {
    println(j)
}
```

We'll get the following output:

```
> Heath Ledger
> Joaquin Phoenix
> Jack Nicholson
```

You'll see this loop many more times in this book, as it's very useful.

The for loop

While in some languages `for-each` and `for` loops are two completely different constructs, in Kotlin a `for` loop is simply a `for-each` loop over a range.

To understand it better, let's look at a `for` loop that prints all the single-digit numbers:

```
for (i in 0..9) {
    println(i)
}
```

This doesn't look anything like a Java `for` loop and may remind you more of Python. The two dots are called a **range operator**.

If you run this code, you will notice that this loop is inclusive. It prints all the numbers, including `9`. This is similar to the following Java code:

```
for (int i = 0; i <= 9; i++)
```

If you want your range to be exclusive and not to include the last element, you can use the `until` function:

```
for (i in 0 until 10) {
    println("for until $i")
// Same output as the previous
        loop
}
```

If you'd like to print the numbers in reverse order, you can use the `downTo` function:

```
for (i in 9 downTo 0) {
    println("for downTo $i") // 9, 8, 7...
}
```

It may seem confusing that `until` and `downTo` are called functions, although they look more like operators. This is another interesting Kotlin feature called **infix call**, which will be discussed later.

The while loop

There are no changes to the **`while`** loop functionality compared to some other languages, so we'll cover them very briefly:

```
var x = 0
while (x < 10) {
    x++
    println("while $x")
}
```

This will print numbers from **`1`** to **`10`**. Note that we are forced to define **`x`** as **`var`**. The lesser-used **`do while`** loop is also present in the language:

```
var x = 5
do {
    println("do while $x")
    x--
} while (x > 0)
```

Most probably, you won't be using the **`while`** loop and especially the **`do while`** loop much in Kotlin. In the following chapters, we'll discuss much more idiomatic ways to do this.

Classes and inheritance

Although Kotlin is a multi-paradigm language, it has a strong affinity to the Java programming language, which is based on classes. Keeping Java and JVM interoperability in mind, it's no wonder that Kotlin also has the notion of classes and classical inheritance.

In this section, we'll cover the syntax for declaring classes, interfaces, abstract classes, and data classes.

Classes

A **class** is a collection of data, called properties, and methods. To declare a class, we use the **`class`** keyword, exactly like Java.

Let's imagine we're building a video game. We can define a class to represent the player as follows:

```
class Player {
}
```

The instantiation of a class simply looks like this:

```
val player = Player()
```

Note that there's no **`new`** keyword in Kotlin. The Kotlin compiler knows that we want to create a new instance of that class by the **round brackets** after the class name.

If the class has no body, as in this simple example, we can omit the curly braces:

```
class Player // Totally fine
```

Classes without any functions or properties aren't particularly useful, but we'll explore in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, why this syntax exists and how it is consistent with other language features.

Primary constructor

It would be useful for the player to be able to specify their name during creation. In order to do that, let's add a primary constructor to our class:

```
class Player(name: String)
```

Now, this declaration won't work anymore:

```
val player = Player()
```

Also, we'll have to provide a name for every new player we instantiate:

```
val player = Player("Roland")
```

We'll return to constructors soon enough. But for now, let's discuss properties.

Properties

In Java, we are used to the concept of getters and setters. If we were to write a class representing a player in a game in Kotlin using Java idioms, it may have looked like this:

```
class Player(name: String) {
    private var name: String = name
    fun getName(): String {
        return name
    }
    fun setName(name: String) {
        this.name = name;
    }
}
```

If we want to get a player's name, we invoke the **`getName()`** method. If we want to change a player's name, we invoke the **`setName()`** method. That's quite simple to follow but very verbose.

It is the first time we see the **`this`** keyword in Kotlin, so let's quickly explain what it means. Similar to many other languages, **`this`** holds the reference to the current object of that class. In our case, it points to the instance of a **`Player`** class.

Why don't we write our classes like that, though?

```
class Player {
    var name: String = ""
}
```

Seems like this approach has lots of benefits. It is much less verbose for sure. Reading a person's name is now much shorter – **`player.name`**.

Also, changing the name is much more intuitive – **`player.name = "Alex";`**.

But by doing so, we lost a lot of control over our object. We cannot make **`Player`** immutable, for example. If we want everybody to be able to read the player's name, they'll also be able to change it at any point in time. This is a significant problem if we want to change that code later. With a setter, we can control that, but not with a public field.

Kotlin properties provide a solution for all those problems. Let's look at the following class definition:

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

Note that this is almost the same as the example from the *Primary constructor* section, but now **`name`** has a **`val`** modifier.

This may look the same as the **`PublicPerson`** Java example, with all its problems. But actually, this implementation is similar to **`ImmutablePerson`**, with all its benefits.

How is that possible? Behind the scenes, Kotlin will generate a member and a getter with the same name for our convenience. We can set the property value in the constructor and then access it using its name:

```
val player = Player("Alex")
println(player.name)
```

Trying to change the name of our **`Player`** will result in an error, though:

```
player.name = "Alexey" // value cannot be reassigned
```

Since we defined this property as a value, it is read-only. To be able to change a property, we need to define it as mutable. Prefixing a constructor parameter with **`var`** will automatically generate both a getter and a setter:

```
class Player(val name: String, var score: Int)
```

If we don't want the ability to provide the value at construction time, we can move the property inside the class body:

```
class Player(val name: String) {
    var score: Int = 0
}
```

Note that now we must also provide a default value for that property, since it cannot be simply **`null`**.

Custom setters and getters

Although we can set a score now easily, its value may be invalid. Take the following example:

```
player.score = -10
```

If we want to have a mutable property with some validations, we need to define an explicit setter for it, using **`set`** syntax:

```
class Player(val name: String) {
    var score: Int = 0
        set(value) {
              field = if (value >= 0) {
                  value
              } else {
                  0
              }
         }
}
```

Here, **`value`** is the new value of the property and **`field`** is its current value. If our new value is negative, we decide to use a default value.

Coming from Java, you may be tempted to write the following code in your setter instead:

```
set(value) {
    this.score = if (value >= 0) value else 0
}
```

But, in Kotlin, this will create an infinite recursion. You must remember that Kotlin generates a setter for mutable properties. So, the previous code will be translated to something like this:

```
// This is a pseudocode, not real Kotlin code!
...
fun setValue(value: Int) {
    setValue(value) // Infinite recursion!
}
...
```

For that reason, we use the **`field`** identifier, which is provided automatically.

In a similar manner, we can declare a custom getter:

```
class Player(name: String) {
    val name = name
        get() = field.toUpperCase()
}
```

First, we save a value received as a constructor argument into a field with the same name. Then, we define a custom getter that will convert all characters in this property to uppercase:

```
println(player.name)
```

We'll get this as our output:

```
> ALEX
```

Interfaces

You are probably already familiar with the concept of **interfaces** from other languages. But let's quickly recap.

In typed languages, interfaces provide a way to define behavior that some class will have to implement. The keyword to define an interface is simply **`interface`**.

Let's now define an interface for rolling a die:

```
interface DiceRoller {
    fun rollDice(): Int
}
```

To implement the interface, a class specifies its name after a colon. There's no **`implement`** keyword in Kotlin.

```
import kotlin.random.*
class Player(...) : DiceRoller
{
    ...
    fun rollDice() = Random.nextInt(0, 6)
}
```

This is also the first time we see the **`import`** keyword. As the name implies, it allows us to import another package, such as **`kotlin.random`**, from the Kotlin standard library.

Interfaces in Kotlin also support default functions. If a function doesn't rely on any state, such as this function that simply rolls a random number between **`0`** and **`5`**, we can move it into the interface:

```
interface DiceRoller {
    fun rollDice() = Random.nextInt(0, 6)
}
```

Abstract classes

Abstract classes, another concept familiar to many, are similar to interfaces in that they cannot be instantiated directly. Another class must extend them first. The difference is that unlike **`interface`**, an abstract class can contain state.

Let's create an abstract class that is able to move our player on the board or, for the sake of simplicity, just store the new coordinates:

```
abstract class Moveable() {
    private var x: Int = 0
    private var y: Int = 0
    fun move(x: Int, y: Int) {
        this.x = x
        this.y = y
    }
}
```

Any class that implements **`Moveable`** will inherit a **`move()`** function as well.

Now, let's discuss in some more detail the **`private`** keyword you see here for the first time.

Visibility modifiers

We mentioned the **`private`** keyword earlier in this chapter but didn't have a chance to explain it. The **`private`** properties or functions are only accessible to the class that declared them – **`Moveable`**, in this case.

The default visibility of classes and properties is public, so there is no need to use the **`public`** keyword all the time.

In order to extend an abstract class, we simply put its name after a colon. There's also no **`extends`** keyword in Kotlin.

```
class ActivePlayer(name: String) : Moveable(),
DiceRoller {
...
}
```

How would you be able to differentiate between an abstract class and an interface, then?

An abstract class has round brackets after its name to indicate that it has a constructor. In the upcoming chapters, we'll see some uses of that syntax.

Inheritance

Apart from extending abstract classes, we can also extend regular classes as well.

Let's try to extend our **`Player`** class using the same syntax we used for an abstract class. We will attempt to create a **`ConfusedPlayer`** class, that is, a player that when given (*x* and *y*) moves to (*y* and *x*) instead.

First, let's just create a class that inherits from **`Player`**:

```
class ConfusedPlayer(name: String ):
ActivePlayer(name)
```

Here, you can see the reason for round brackets even in abstract classes. This allows passing arguments to the parent class constructor. This is similar to using the **`super`** keyword in Java.

Surprisingly, this doesn't compile. The reason for this is that all classes in Kotlin are final by default and cannot be inherited from.

To allow other classes to inherit from them, we need to declare them **`open`**:

```
open class ActivePlayer (...) : Moveable(),
DiceRoller {
...
}
```

Let's now try and override the **`move`** method now:

```
class ConfusedPlayer(name : String): Player(name) {
    // move() must be declared open
    override fun move(x: Int, y: Int) {
        this.x = y // must be declared protected
        this.y = x // must be declared protected
    }
}
```

Overriding allows us to redefine the behavior of a function from a parent class. Whereas in Java, `@Override` is an optional annotation, in Kotlin `override` is a mandatory keyword. You cannot hide supertype methods, and code that doesn't use `override` explicitly won't compile.

There are two other problems that we introduced in that piece of code. First, we cannot override a method that is not declared `open` as well. Second, we cannot modify the coordinates of our player from a child class since both coordinates are `private`.

Let's use the `protected` visibility modifier the makes the properties accessible to child classes and mark the function as `open` to be able to override it:

```
abstract class Moveable() {
    protected var x: Int = 0
    protected var y: Int = 0
    open fun move(x: Int, y: Int) {
        this.x = x
        this.y = y
    }
}
```

Now, both of the problems are fixed. You also see the `protected` keyword here for the first time. Similar to Java, this visibility modifier makes a property or a method visible only to the class itself and to its subclasses.

Data classes

Remember that Kotlin is all about productiveness. One of the most common tasks for Java developers is to create yet another **Plain Old Java Object** (**POJO**). If you're not familiar with POJO, it is basically an object that only has getters, setters, and implementation of `equals` or `hashCode` methods. This task is so common that Kotlin has it built into the language. It's called a **data class**.

Let's take a look at the following example:

```
data class User(val username: String, private val
  password: String)
```

This will generate us a class with two getters and no setters (note the **`val`** part), which will also implement **`equals`**, **`hashCode`**, and **`clone`** functions in the correct way.

The introduction of **`data`** classes is one of the most significant improvements in reducing the amount of boilerplate in the Kotlin language. Just like the regular classes, **`data`** classes can have their own functions:

```
data class User(val username: String, private val
  password: String) {
    fun hidePassword() = "*".repeat(password.length)
}
val user = User("Alexey", "abcd1234")
println(user.hidePassword()) // ********
```

Compared to regular classes, the main limitation of **`data`** classes is that they are always **`final`**, meaning that no other class can inherit from them. But it's a small price to pay to have **`equals`** and **`hashCode`** functions generate automatically.

Kotlin data classes versus Java records

Learning from Kotlin, Java 15 introduced the notion of **records**. Here is how we can represent the same data as a Java **`record`**:

```
public record User(String username, String password)
{}
```

Both syntaxes are pretty concise. *Are there any differences, though?*

- Kotlin **`data`** classes a have **`copy()`** function that records lack. We'll cover it in *Chapter 2*, *Working with Creational Patterns*, while discussing the **prototype** design pattern.

- In a record, all properties must be **final**, or, in Kotlin terms, records support only values and not variables.
- The **data** classes can inherit from other classes, while records don't allow that.

To summarize, **data** classes are superior to records in many ways. But both are great features of the respective languages. And since Kotlin is built with interoperability in mind, you can also easily mark a **data** class as a record to be accessible from Java:

```
@JvmRecord
data class User(val username: String, val password:
String)
```

Extension functions

The last feature we'll cover in this chapter before moving on is **extension functions**. Sometimes, you may want to extend the functionality of a class that is declared **final**. For example, you would like to have a string that has the **hidePassword()** function from the previous section.

One way to achieve that is to declare a class that wraps the string for us:

```
data class Password(val password: String) {
    fun hidePassword() = "*".repeat(password.length)
}
```

This solution is quite wasteful, though. It adds another level of indirection.

In Kotlin, there's a better way to implement this.

To extend a class without inheriting from it, we can prefix the function name with the name of the class we'd like to extend:

```
fun String.hidePassword() = "*".repeat(this.length)
```

This looks almost like a regular top-level function declaration, but with one crucial change – before the function name comes a class name. That class is called a **method receiver**.

Inside the function body, `this` will refer to any `String` class that the function was invoked on.

Now, let's declare a regular string and try to invoke this new function on it:

```
val password: String = "secretpassword"
println("Password: ${password.hidePassword()}")
```

This prints the following:

```
> Password: **************
```

What black magic is this? We managed to add a function to a `final` class, something that technically should be impossible.

This is another feature of the Kotlin compiler, one among many. This extension function will be compiled to the following code:

```
// This is not real Kotlin
fun hidePassword(this: String) {
    "*".repeat(this.length)
}
```

You can see that, in fact, this is a regular top-level function. Its first argument is an instance of the class that we extend. This also might remind you of how methods on structs in **Go** work.

The code that prints the masked password will be adapted accordingly:

```
val password: String = "secretpassword"
println("Password: ${hidePassword(password)}")
```

For that reason, the extension functions cannot override the member function of the class, or access its `private` or `protected` properties.

Introduction to design patterns

Now that we are a bit more familiar with basic Kotlin syntax, we can move on to discuss what design patterns are all about.

What are design patterns?

There are different misconceptions surrounding design patterns. In general, they are as follows:

- Design patterns are just missing language features.
- Design patterns are not necessary in a dynamic language.
- Design patterns are only relevant to object-oriented languages.
- Design patterns are only used in enterprises.

Actually, design patterns are just a proven way to solve a common problem. As a concept, they are not limited to a specific programming language (Java), nor to a family of languages (the C family, for example), nor are they limited to programming in general. You may have even heard of design patterns in software architecture, which discuss how different systems can efficiently communicate with each other. There are service-oriented architectural patterns, which you may know as **Service-Oriented Architecture** (**SOA**), and microservice design patterns that evolved from SOA and emerged over the past few years. The future will, for sure, bring us even more design pattern families.

Even in the physical world, outside software development, we're surrounded by design patterns and commonly accepted solutions to a particular problem. Let's look at an example.

Design patterns in real life

Did you ride an elevator lately? Was there a mirror on the wall of the elevator? Why is that? How did you feel when you last rode an elevator that had no mirror and no glass walls?

The main reason we commonly have mirrors in our elevators is to solve a frequent problem – riding in an elevator is boring. We could put in a picture. But a picture would also get boring after a while, if you rode the same elevator at least twice a day. Cheap, but not much of an improvement.

We could put in a TV screen, as some do. But it makes the elevator more expensive. And it also requires a lot of maintenance. We need to put some content on the screen to make it not too repetitive. So, either there's a person whose responsibility is to renew the content once in a while or a third-party company that does it for us. We'll also have to handle different problems that may occur with screen hardware and the software behind it. Seeing the *blue screen of death* is amusing, of course, but only mildly.

Some architects even go for putting elevator shafts on the building exterior and making part of the walls transparent. This may provide some exciting views. But this solution also requires maintenance (dirty windows don't make for the best view) and a lot of architectural planning.

So, we put in a mirror. You get to watch an attractive person even if you ride alone. Some studies indicate that we find ourselves more attractive than we are, anyway. Maybe you get a chance to review your appearances one last time before that important meeting. Mirrors visually expand the visual space and make the entire trip less claustrophobic or less awkward if it's the start of a day and the elevator is really crowded.

Design process

Let's try and understand what we did just now.

We didn't invent mirrors in elevators. We've seen them thousands of times. But we formalized the problem (riding in an elevator is boring) and discussed alternative solutions (TV screens and glass walls) and the benefits of the commonly used solution (solves the problem and is easy to implement). That's what design patterns are all about.

The basic steps of the design process are as follows:

1. Define exactly what the current problem is.
2. Consider different alternatives, based on the pros and cons.
3. Choose the solution that solves the problem while best fitting your specific constraints.

Why use design patterns in Kotlin?

Kotlin comes to solve the real-world problems of today. In the following chapters, we will discuss both the *design patterns* first introduced by the *Gang of Four* back in 1994, as well as design patterns that emerged from the functional programming paradigm and the design patterns that we use to handle concurrency in our applications.

You'll find that some of the design patterns are so common or useful that they're already built into the language as reserved keywords or standard functions. Some of them will need to combine a set of language features. And some are not so useful anymore, since the world has moved forward, and other patterns are replacing them.

But in any case, familiarity with design patterns and best practices expands your *developer toolbox* and creates a shared vocabulary between you and your colleagues.

Summary

In this chapter, we covered the main goals of the Kotlin programming language. We learned how variables are declared, the basic types, `null` safety, and type inference. We observed how program flow is controlled by commands such as `if`, `when`, `for`, and `while`, and we also took a look at the different keywords used to define classes and interfaces: class, interface, **data** class, and **abstract** class. We learned how to construct new classes and how to implement interfaces and inherit from other classes. Finally, we covered what design patterns are suitable for and why we need them in Kotlin.

Now, you should be able to write simple programs in Kotlin that are pragmatic and type-safe. There are many more aspects of the language we need to discuss. We'll cover them in later chapters once we need to apply them.

In the next chapter, we'll discuss the first of the three design pattern families – creation patterns.

Questions

1. What's the difference between `var` and `val` in Kotlin?
2. How do you extend a class in Kotlin?
3. How do you add functionality to a `final` class?

Support | Sign Out

©2022 O'REILLY MEDIA, INC. TERMS OF SERVICE | PRIVACY POLICY