Table of Contents

Introduction

Chapter 1
First Program with Kotlin

Installation of Kotlin

In order to compile and execute programs written in Kotlin, you need a compiler. To get the latest version of the Kotlin compiler, download it at the following URL: https://github.com/JetBrains/kotlin/releases/latest.

Once the .zip file has been downloaded, extract it in the folder of your choice. Then, you need to add the path to the `bin` folder it contains in the `PATH` environment variable of your system. How to do this depends on which operating system you use.

Windows

- In the Control Panel, choose *Small Icons* right after *View by* in the top-right corner
- Click on *System*
- On the left, click on *Advanced system settings…*
- In the dialog box, click on the `Environment Variables…` button
- In the bottom frame, click on the line for which the value of the Variable column is `Path`
- Click on the `Edit…` button
- Add the path to the `bin` folder in a new line
- Click `OK` three times to close the three open dialog boxes

macOS

- Open a terminal
- Type `sudo nano /etc/paths`, then hit the Enter key
- Add the path to the `bin` folder in a new line
- Type `Ctrl+O` to save, then `Ctrl+X` to quit

Linux

How to complete the Kotlin installation will vary depending on your distribution. But if you are a Linux user, chances are good that you know how to edit the `PATH` environment variable by yourself.

Hello, World!

Source Code

A **Hello, World!** is a simple program that aims to provide a first foray into a programming language. It's a basic program that displays `Hello, World!` on your screen. The main purpose of a **Hello, World!** is to keep your first experience with a programming language as simple as possible.

Open a text editor (such as the excellent program Atom, for example), and write the following content in a new file. Then, save the file with the name `HelloWorld.kt`.

HelloWorld.kt

```
fun main(args: Array<String>) {
   println("Hello, World!")
}
```

From Source Code to a Running Program

You'll need to take three steps to run your new Kotlin program.

Write source code

That's what we just did in the text editor. It is all about writing a program in a language that a developer can read. What we wrote is simple code that can be read and modified easily enough by any developer.

Compiling

This step is about translating your source code into a file that your computer can understand. For some languages, such as PHP or JavaScript, you don't need to compile source code. The system will be able to read and interpret your program directly. Therefore, it is not the best in terms of performance.

As humans, we are able to write code that a computer can understand right away. The problem with it is that it will be complex to read and to modify by a human.

HelloWorld.exe

```
4D 5A 3B 00 01 00 00 00 02 00 00 01 FF FF 02 00
00 10 00 00 00 00 00 00 1C 00 00 00 00 00 00 00
0E 1F B4 09 BA 0E 00 CD 21 B8 00 4C CD 21 48 65
6C 6C 6F 20 57 6F 72 6C 64 21 24
```

Take a look at the program above: there is almost no chance that you understand anything about it. It's an hexadecimal transcription of a `Hello, World!` for 32-bit DOS systems. You can probably see that if we write in the *"natural language"* for the system, it becomes very hard to understand or modify this message.

So the compiler is the tool that will generate a file that is easy to understand for the system, from one or more files that are easy to understand for a human.

System and runtime environment

We just talked about code that is easy to understand for the system. It is important to note that, in this case, the "system" is the environment in which the program is running. It may be the operating system or, as is the case for Java (and also for Kotlin, which is based on Java), a dedicated runtime environment.

The advantage of a dedicated runtime environment like this is that you can write programs and share binaries regardless of the operating system on which it will be run.

For example, a `.exe` binary can be run on Windows, but if you try to run it on macOS, this will never work. But a Java or Kotlin binary file, which is a `.class` file, can be run on Windows and macOS as well as on Android.

Run

The final step is running the program. This step will produce the final result.

Compilation with Kotlin

To compile a program with Kotlin, you just need to run the following command in a terminal:

```
kotlinc HelloWorld.kt
```

This command will create a file named `HelloWorldKt.class`. It is a Kotlin binary file.

Run with Kotlin

To run a Kotlin program that has been compiled, you just need to run the following command:

```
kotlin HelloWorldKt
```

This will produce the following result:

```
Hello, World!
```

Congratulations. You just wrote a source code, compiled it, and ran your first Kotlin application.

More Information about Source Code

Let's go back to our source code. This time, we will focus on understanding exactly what it does.

HelloWorld.kt

```
fun main(args: Array<String>) {
   println("Hello, World!")
}
```

Functions and Instructions

The `fun` keyword let Kotlin know that we will define a function. When talking about programming, a function is a group with one or more instructions to be run in a specific order.

An instruction is just a line calling an action to be performed by the computer. This action may be a system action (create a file, display text on screen, etc.), or a call to a function.

Right after the `fun` keyword comes the name of the function, which is `main` in the current file. Because this function is named `main`, Kotlin knows that this is the function it has to call when we run the `HelloWorldKt` program. Actually, we can write many functions in a single program. If we do so, Kotlin needs to know which one should be called first.

Between parentheses, there are arguments of our function. Here, we only have one argument: `args: Array<String>`. The arguments are the data that the function will interact with. In the current function, those data are useless. So we will not go deep into the arguments of this `main` function, but they are required here. If we wrote our `main` function like this: `fun main()`, there would have been a bug while running—even though writing it like this is absolutely correct and is a valid `main` function. It has more to do with a bug with the specific `main` function, which is the first one called when running a program.

Then, we have an opening bracket and a closing one at the end of the file. Those brackets are here to delimit the instructions for a function. So all the instructions between those brackets are instructions for the main function.

Println

Then, we call the `println` function. The main purpose of this function is to print text on a new line. The argument of this function is the string `"Hello, World!"` So we see how useful arguments are here. In this case, the argument is the data that will be printed by `println`.

Conclusion

Now we've seen how to write a source code, compile it, and run the binaries with Kotlin. We also know what functions, instructions, and arguments are.

In case you're feeling a little fuzzy on these terms: in the beginning of this book, there will be a lot of definitions that you may find abstract and that you may find difficult to understand. This is absolutely normal and should not discourage you.

Yes, those terms need definitions when we use them the first time, but you don't need to learn them by heart to continue reading (and understanding) the book.

It is absolutely not required to know the definitions of those terms by heart. In fact, I had to go to Wikipedia to get a better idea of how to define those terms. Once you have developed some programs, you will find that you know what a function is, although you may not be able to define what it is precisely. You will just have a clear idea about what it is and what it is not (and how to apply that knowledge), and that is the important part.

Exercises

Each chapter will end with one or more exercises, and each of these will include a word to describe how difficult it is: easy, medium, or difficult. It is not about how complex it is (because complexity is specific to each person), but is more of a scale to help you know what to expect:

- **Easy**: Everything you need to complete the exercise is in the chapter. You just have to apply the theory to this specific task.
- **Medium**: You can complete this exercise with the content of the chapter, but you will need to experiment, try things, and get things wrong in order to find the way to do it. This is because everything you have to do is not necessarily in this chapter.
- **Difficult**: You will not be able to complete the exercise with the content of the chapter alone. You will have to check on the Internet or use other sources to help you complete the task.

Of course, I could write an exhaustive book with every piece of information you need to complete these exercises. But the aim is to help you learn to face these problems. Once more, the main purpose of this book is to enable you to become an entry-level developer. And, a developer will

spend all his days trying, helping coworkers, getting help from coworkers, and finding solutions to problems he encounters on the Internet. Since you have to get used to this process as soon as possible, let's do it … right now:

Exercise 1 - Medium

Go back to the `HelloWorld.kt` program, and edit it so that:

- It consists of two functions, `main` and `displayHelloWorld`.
  - `displayHelloWorld`, without any argument, will be a function that displays `Hello, World!`
  - `main` will only call `displayHelloWorld`.

Exercice 2 - Difficile

We saw that when compiling a program, a file named `HelloWorldKt.class` is created in the current folder. Find a way to compile it so `HelloWorld.kt` will be placed in a folder called `exercise`, and use only `kotlinc` command. In other words, don't create the `exercise` folder any way but by using `kotlinc` command.

Chapter 2
Variables, Types, Constants, and Operators

Definition of a Variable

A variable is a value paired with an associated symbolic name. If you've used mathematics before, you have already used variables. When you are asked to give the formula of the circumference of a circle, you answer π times diameter. When you say that, you associate the *diameter* word with a value.

Variable Types

In computer programming, a value is a storage location. So you will have to make the best use of the storage location in order to optimize the performance of an application.

Let's say that you want to make a quiz in which you let the user guess a number between 1 and 100. The number may be stored on one byte (28, so 256 possible values). So you will associate this number to a value that can be stored on one byte, which optimizes the amount of memory consumed by your application.

Now, let's say you would like to associate each person in a group to a number, and let's say there are 2,500 to 3,000 people to this group. You will easily guess you can not store each person's number on one byte without running out of storage. You will need a type that can store more data.

Numbers

That is why Kotlin comes with 8 variable types in which you can store numbers:

- `Byte` : a value from -128 to +127 stored on one byte
- `Short` : a value from -32,768 to +32,767 stored on two bytes
- `Int` : a value from -231 to +231-1 stored on four bytes
- `Long` : a value from -263 to +263-1 stored on eight bytes
- `Float` : a value stored on sixteen bytes in which you can store more precise decimal numbers
- `Double` : a value stored on thirty-two bytes in which you can store the most precise decimal numbers

Be careful: `Float` and `Double` must always be considered approximate values (with `Double` being more precise) but never as exact values.

Characters

That is OK for numbers, but you may also want to store characters in variables. A character is a sign that can be printed to the screen, such as a letter, a tab, a line break, or a punctuation mark.

- `Char` : a value to store characters stored on two bytes

Boolean

Finally, the boolean type can store only two values: true and false.

- `Boolean` : a type with which to store a binary value, stored on a very small undefined amount of memory (less than or equal to one byte).

Object

Variables can also store object instances. But object-oriented programming will be covered later in this book.

Notation

To define a variable in Kotlin we use the `var` keyword, followed by the name of the variable, a colon, its type, the `=` sign, and its value. This is called a mutable variable because its value may change.

So, the definition of a `Double` variable named `diameter` would be written as follows:

```
var diameter: Double = 3.78
```

Definition of an Immutable Variable

An immutable variable is a variable for which the value will not change. You initialize its value, and then you cannot edit it. If you try to edit the value of an immutable variable, the compiler will not compile your program.

Aside from the fact that its value cannot be edited, everything that has been said about mutable variables is also true about immutable variables.

Notation

To define an immutable variable in Kotlin, we use the same notation we do for mutable variables —except for the `var` keyword, which becomes `val` .

```
val pi: Double = 3.14
```

Definition of a Constant

A constant is an identifier that will be replaced by its value by the preprocessor of the compiler, before the compiling begins.

So far, we have seen all steps from source code to final result, including compilation.

Right before compilation, there is a preprocessor that will edit your source code. This preprocessor will replace all constant identifiers with their values.

This means you can only define a constant value with an explicit value. You cannot set to a constant the value returned by a function (we will see that later in this chapter) but only a number or a string…

In order to produce a cleaner code, it is preferable to use a constant. However:

- If it is not possible to use a constant, then use an immutable variable.
- If it is not possible to use an immutable variable, then use a mutable variable.

Notation

The notation used to define a constant in Kotlin is the same as it is for an immutable variable, except that the `val` keyword is preceded by the `const` keyword.

```
const val PI: Double = 3.14
```

Definition of an Argument

An argument is a variable passed to a function so the function can interact with it. For example, when we develop a function to calculate the circumference of a circle, we need to know the diameter of the circle. This value may be passed as an argument to the function.

Notation

An argument is written between two parentheses in the definition of a function. We write the name of the argument, followed by a colon, followed by its type. A function that will calculate the circumference of a circle will be written as follows:

```
fun circumference(diameter: Double){

}
```

That is how you write a function with one argument. But you may pass more arguments to a single function. If you do this, arguments must be separated by a comma. So if we write a function calculating the perimeter of a rectangle, we will have to use the height and the width as arguments. Therefore, the function will be written as follows:

```
fun rectanglePerimeter(width: Double, height: Double){

}
```

Return Value of a Function

A function may return a value. In our first **Hello, World!** sample, the `main` function did not return a value. But if we write a function calculating the circumference of a circle, we may want the function to return the value of the circumference. To specify that a function should return a value, right after closing the parentheses, we add a semicolon and then its type. So a function calculating the circumference of a circle will be written as follows:

```
fun circumference(diameter: Double): Double{

}
```

A function can return only one value, and that value can only be of one specified type.

Of course, you can assign the return value of a function to a variable. This will be written as follows:

```
val diameter : Double = 3.78
val circumference : Double = circumference(diameter)
```

Implicit Types and Literals

We just saw how to define a variable:

```
val someValue: Int = 4
```

In that specific case, we don't have to specify the `Int` variable type. We can simply write it as follows:

```
val someValue = 4
```

This notation is correct and is equivalent to the first one. Be careful: even if the type is not explicitly set, 4 is still an `Int` value. To be sure that this is the case, you may write the following program:

TypeError.kt

```
fun main(args: Array<String>) {
   var someValue = 4
   someValue = 5.56
   println(someValue)
}
```

When you try to compile the program, you will get the following error:

```
TypeError.kt:3:17: error: the floating-point literal does not conform to the expected type Int
    someValue = 5.56
                ^
```

Kotlin tells you that it can not set a decimal number to a variable of type `Int`.

So, which type will be set to a variable when we do not explicitly specify its type?

We can find out by writing another little Kotlin program:

VarTypes.kt

```
fun main(args: Array<String>) {
   val someValue = 4
   val someValue2 = 4.56
   val someValue3 = 'c'

   println(someValue.javaClass.kotlin.qualifiedName)
   println(someValue2.javaClass.kotlin.qualifiedName)
   println(someValue3.javaClass.kotlin.qualifiedName)
}
```

This program uses parts of Kotlin that we will see later in this book. Just copy and paste this code as it is; it is not really important that you understand everything about it right now.

When compiling and then running the program, we get the following result:

```
kotlin.Int
kotlin.Double
kotlin.Char
```

So we now know that, when we write an integer number, the `Int` type will be set. When we write a decimal number, the `Double` type will be set; and when we write a character in single quotes, the `Char` type will be set.

But how do we indicate that a variable is of the `Float` type or the `Long` type without specifying its type explicitly?

We may use a character at the end of the value to specify the type of variable it is:

VarTypesFull.kt

```
fun main(args: Array<String>) {
    val someValue = 4
    val someValue2 = 4L
    val someValue3 = 4.56
    val someValue4 = 4.56F
    val someValue5 = 'c'

    println(someValue.javaClass.kotlin.qualifiedName)
    println(someValue2.javaClass.kotlin.qualifiedName)
    println(someValue3.javaClass.kotlin.qualifiedName)
    println(someValue4.javaClass.kotlin.qualifiedName)
    println(someValue5.javaClass.kotlin.qualifiedName)
}
```

When compiling and then running this program, we get the following result:

```
kotlin.Int
kotlin.Long
kotlin.Double
kotlin.Float
kotlin.Char
```

So what we see here is that by adding an `L` to the end of an integer number, we set its type to `Long`. By adding an `F` to the end of a decimal number, we set its type to `Float`. These are the only two types for which we are able to use those shortcuts.

To define a variable as `Short` or `Byte`, we need to use the explicit definition.

Integers Notation

In order to produce a more readable and/or comprehensive code, you may use some different notations for integers.

Underscores

For very large numbers, you may use underscores to separate digits by groups in a number. This helps make the number more readable.

```
val complexGigaByte = 1073741824
val readableGigaByte = 1_073_741_824
```

Both declarations are equivalent, but you may notice that the second one is more readable than the first.

Binary Notation

Sometimes, you may want to write a number using its binary notation. It's possible to do this by adding the 0b prefix to the number. Underscores may also be used:

```
val readableGigaByte = 1_073_741_824
val binaryGigaByte = 0b1000_0000_0000_0000_0000_0000_0000_000
```

Both declarations are equivalent.

Hexadecimal Notation

You may also want to write numbers using their hexadecimal notation. It is possible to do this by adding the 0x prefix to those numbers:

```
val readableGigaByte = 1_073_741_824
val hexadecimalGigaByte = 0x4000_0000
```

Operators

Assignment

This is the only operator we have already used. You can use an `=` sign to assign a value to a variable:

```
val three = 3
```

Mathematical Operators

It is possible to use mathematical operators between two numeric values or variables:

Operators.kt

```
fun main(args: Array<String>) {
   val add = 2_000_000_000+2_000_000_000
   val multiply = 3*7
   val subtract = 7-4
   val divide = 7/3
   val modulo = 7%3

   println(add)
   println(multiply)
   println(subtract)
   println(divide)
   println(modulo)
}
```

The modulo operation returns the remainder in the euclidean division.

When compiling and then running this program, we get the following result:

```
-294967296
21
3
2
1
```

As you can see, the sum of two billions plus two billions produces a negative result. You may also notice that the result of 7/3 is 2. This is because an operation between two variables of the same type produces a result of the same type. Here, an addition of two `Int`, or a division between two `Int`, produces an `Int`.

Now we understand why 7/3 returns 2 but not why the sum is producing a negative result. We already said that an `Int` cannot store a value higher than 2^{31}-1, or 2,147,483,647. If we are going above this value, nothing reliable can be expected from an `Int` variable.

So what about operations between two different types? The type stored on more bytes will be the type of the result. The only exception is the `Char` type, which will still be a `Char` even if we add a value that is stored on more bytes. So, to get the expected result, we can write the following program:

OperatorsCorrect.kt

```
fun main(args: Array<String>) {
    val byte:Byte = 7
    val short:Short = 5

    val add = 2_000_000_000+2_000_000_000L
    val multiply = 3*7
    val subtract = 7-4
    val divide = 7/3.0
    val modulo = 7%3
    val add2 = byte+2_000_000_000
    val add3 = short+2_000_000_000
    val add4 = 'c'+3

    println(add)
    println(multiply)
    println(subtract)
    println(divide)
    println(modulo)
    println(add2)
    println(add3)
    println(add4)
}
```

When compiling and then running the program, we get the following result:

```
4000000000
21
3
2.3333333333333335
1
2000000007
2000000005
f
```

Augmented Assignments

You may need to write the following code:

```
var value = 3
value = value + 10
value = value - 1
value = value / 3
value = value * 4
value = value % 5
```

When the variable is both the assigned value and the first part of a mathematical operation, you can write the code as follows:

```
var value = 3
value += 10
value -= 1
value /= 3
value *= 4
value %= 5
```

The two codes are equivalent.

About Strings

You may use the `+` sign in a string to concatenate the result:

```
val HelloWorld100 = "Hello "+"World! "+100;
```

The value of the `HelloWorld100` variable is now `"Hello World! 100"`.

Let's Practice

In your text editor, open a new file that you will name `Circumference.kt`:

Circumference.kt

```
fun main(args: Array<String>) {
   var diameter: Double = 3.78
   var circumference = circumference(diameter)
   println(circumference)
   diameter = 1.72
   circumference = circumference(diameter)
   println(circumference)
}

fun circumference(diameter: Double): Double{
   return 3.14*diameter
}
```

This program returns the circumference of a circle regarding its diameter. When you compile and then run this program, you will get the following result:

```
47.4768
22.2312
```

We could be happy with that, but there is just one little thing missing.

Code Cleaning

The content of the `main` function is difficult to approach. It needs to be read line by line to clearly understand what it does. But it is possible to make things more clear.

Comments

You may add comments to your code. Comments are a part of code that the compiler will ignore but that may help with the readability of the program.

Single-Line Comments

To write a comment on a single line, you have to add `//`. Everything after that on the line will be ignored by the compiler, except `//`, which is included in a string.

```
var diameter: Double = 3.78 // diameter is now 3.78
println("I am displayed // me too") // but I am not
```

Blocks of Comments

It is also possible to write comments on two or more lines. The comment needs to start with `/*` and to end with `*/`:

```
/*Calculate the circumference of a circle.
The formula in use is pi*diameter
This is much more clear*/
circumference = circumference(diameter)
```

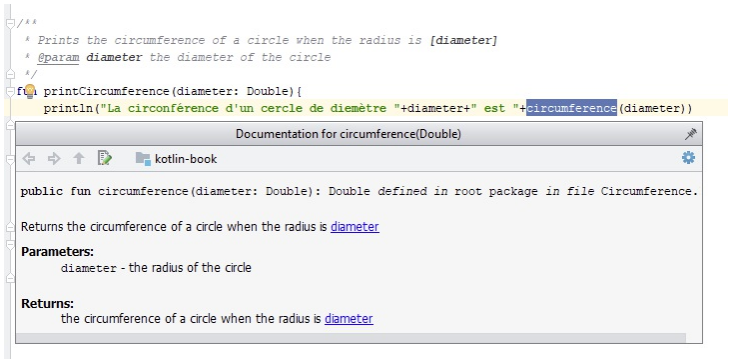
KDoc

KDoc is a special form of comment that is placed before a block and provides details about this block. The advantage with KDoc is that it is known and used by all, so generating or accessing documentation is easier in this format.

We could spend a full chapter on KDoc. But for now we'll take a look at some examples, and then you will become more comfortable with KDoc as you read this book.

```
/**
* Returns the circumference of a circle when diameter is [diameter].
* @param diameter the diameter of the circle
* @return the circumference of a circle when diameter is [diameter]
*/
fun circumference(diameter: Double): Double{
   return 3.14*diameter
}
```

The KDoc block here lets the reader clearly understand what the function does and how arguments are used.



Exemple d'affichage d'une boîte d'aide sur le rôle d'une fonction pour laquelle une documentation au format KDoc a été rédigée dans Android Studio.

English Language

This book was originally written in French. In the French version, all of the programs are written in English. So regardless of your native tongue, there are a few good reasons to write your programs in English:

- All Kotlin functions are in English, just like the `println` function we used. Writing your code in English will keep you from getting lost between two different languages in your code; it will also preserve homogeneity in your code.
- You will use a lot of English websites to ask questions about parts of your code, and you'll also use English websites to copy/paste available code. Again, keeping everything in English will keep you from getting lost and will help people who would like to answer your questions to better understand you.
- Finally, variable identifiers and functions cannot be written using special characters. You can not use the French "é", Spanish "ñ", or German "ö" in identifiers. Because English doesn't have special characters, keeping everything in English will let you write without making spelling errors.

Code Refactoring

As you can see, we repeat the same things two times in our code:

- diameter = value
- circumference = returnValueOfFunction
- display circumference

The problem when duplicating code is that when we want to edit it, fix bugs, or improve it, we will need to do it twice.

Constants

In order to make the code more readable, it is better to take out direct values from the code and to put it into constants instead.

This is mainly because you may make a mistake when typing a string or a number in your code. If you do so, the code may be very difficult to fix (especially with huge projects).

Using constants will make things easier because if you make a mistake writing the name of a constant, the compiler simply won't compile. Instead it will display an error message showing you exactly where the error is.

In order to differentiate constants and variables, we often write the constants' names in uppercase letters.

Let's Practice Again

By following the above advice, we get the following code:

Circumference.kt

```
/** The value of π for calculating the circumference */
const val PI: Double = 3.14

/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printCircumference(1.72)
}

/**
 * Prints the circumference of circle when diameter is [diameter]
 * @param diameter the diameter of circle
 */
fun printCircumference(diameter: Double){
   println(circumference(diameter))
}

/**
 * Returns the circumference of circle when diameter is [diameter]
 * @param diameter the diameter of circle
 * @return the circumference of a circle when diameter is [diameter]
 */
fun circumference(diameter: Double): Double{
   return PI*diameter
}
```

This makes the code easier to read and maintain. Let's say that you now want to display `"circumference("+diameter+")="+circumference(diameter)` . You will have to edit only one part of your code:

Circumference.kt

```
/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+circumference(diameter))
}
```

Code refactoring lets us edit just one part of the code to update two displays.

Exercises

Exercise 1 - Easy

Develop a program equivalent to the one we just developed but which will calculate the perimeter of a square using the length of its side.

Exercise 2 - Easy

Develop a program that, from the values 13 and 5, will display `"13/5 = 2 and the remainder is 3"`.

Chapter 3
Multiple Sources, Packages, and Imports

In this chapter, some definitions will be highlighted because they are incomplete or too imprecise. This is because you would need object-oriented programming knowledge to understand the exact definition. Object-oriented programming will be covered in the next chapter.

By now, you know how to develop functions with variables. So you may want to develop huge programs. You're now able, for example, to develop a project which calculates the perimeters and areas of all possible plane shapes.

Just imagine: quadrilaterals, circles, triangles... and why not pentagons, hexagons, or dodecagons? So source code would become very long and, therefore, would have at least two inconvenient aspects:

- First, it would become very complex to maintain a large amount of source code in a single file.
- Second, if your program starts becoming too large, issues with loading can happen because your computer needs to load all of the functions you developed.

That is exactly why Kotlin provides import and package features to remedy these issues.

Multiple Sources

We will write a program similar to the one we developed in the previous project except that it will be separated into two files:

- The first will contain display functions of the program.
- The second will contain circle calculation functions.

Let's begin:

Geometry.kt

```
/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printCircumference(1.72)
}

/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+circumference(diameter))
}
```

Circle.kt

```
/** The value of π for calculating the circumference of a circle */
const val PI: Double = 3.14

/**
 * Returns the circumference of a circle when the diameter is [diameter]
 * @param diameter the diameter of the circle
 * @return the circumference of a circle when the diameter is [diameter]
 */
fun circumference(diameter: Double): Double{
   return PI*diameter
}
```

Now, execute the following command:

```
kotlinc Geometry.kt Circle.kt -d ./build
```

The `-d` argument, which you'll know if you completed the exercises from Chapter 1, allows us to create a folder in which the binary files will be stored.

This allows us to avoid mixing source files and binary files, which will become important when we start to work with projects containing multiple files.

To run the program, you just need to execute the following command:

```
kotlin -cp ./build GeometryKt
```

The `-cp` argument lets the Kotlin compiler know in which directory it should look for binaries.

You can see that two `.class` binary files have been created. So, we know that Kotlin separated our source files.

Packages

We've seen how to compile a program to determine which sources are in separated files. But it is possible to do more.

Here, the `PI` constant has been added to a file calculating the circumference of a circle. Imagine now that we would like to give our program the ability to convert angles measured in radians to degrees, and vice versa. We will need the `PI` constant for angle calculations as well.

This will force us to create four files in order to not duplicate code as seen in the previous chapter.

Geometry.kt

```
/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printAngleDeg(PI)
}

/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+circumference(diameter))
}

/**
 * Prints the conversion of an angle measured in radian to
 */
fun printAngleDeg(angRad: Double){
   println("toDegrees("+angRad+")="+toDegrees(angRad))
}
```

Circle.kt

```
/**
 * Returns the circumference of a circle when the diameter is [diameter]
 * @param diameter the diameter of the circle
 * @return the circumference of a circle when the diameter is [diameter]
 */
fun circumference(diameter: Double): Double{
   return PI*diameter
}
```

Angle.kt

```
/** Flat angle measured in degrees */
const val DEGREES_FLAT_ANGLE = 180
/** Flat angle measured in radians */
const val RADIANS_FLAT_ANGLE = PI

/**
 * Converts an angle measured in radians to an equivalent angle measured in degrees.
 * @param angrad an angle, in radians
 * @return the measurement of the angle [angrad] in degrees
 */
fun toDegrees(angrad: Double): Double{
   return angrad/RADIANS_FLAT_ANGLE*DEGREES_FLAT_ANGLE
}

/**
 * Converts an angle measured in degrees to an equivalent angle measured in radians.
 * @param angdeg an angle, in degrees
 * @return the measurement of the angle [angdeg] in radians
 */
fun toRadians(angdeg: Double): Double{
   return angdeg/DEGREES_FLAT_ANGLE*RADIANS_FLAT_ANGLE
}
```

Constants.kt

```
/** The value of π for calculating the circumference of a circle */
const val PI: Double = 3.14
```

In this example, we only have four files. But sometimes a project may contains hundreds or even thousands of files. So we do not want to add each file to the compile command.

```
kotlinc ./* -d build
```

The program compiles and runs well. Everything is fine.

Now, imagine we add two more files to our project, one to calculate the perimeter of a square and another to calculate the perimeter of an equilateral triangle.

Square.kt

```
/**
 * Returns the perimeter of a square when the length of a side is [sideLength]
 * @param sideLength the length of a side of the square
 * @return the perimeter of a square when the length of a side is [sideLength]
 */
fun perimeter(sideLength: Double): Double{
   return 4*sideLength
}
```

EquilateralTriangle.kt

```
/**
 * Returns the perimeter of an equilateral triangle when the length of a side is [sideLength]
 * @param sideLength the length of a side of the equilateral triangle
 * @return the perimeter of an equilateral triangle when the length of a side is [sideLength]
 */
fun perimeter(sideLength: Double): Double{
   return 3*sideLength
}
```

Once more, let's compile our program:

macOS and Linux :

```
rm -rf ./build/* && kotlinc ./* -d build
```

Windows :

```
del /S/Q build && kotlinc ./* -d build
```

Now we get an error:

```
EquilateralTriangle.kt:6:1: error: conflicting overloads: public fun perimeter(sideLength: Dou
ble): Double defined in root package in file EquilateralTriangle.kt, public fun perimeter(side
Length: Double): Double defined in root package in file Square.kt
fun perimeter(sideLength: Double): Double{
^
Square.kt:6:1: error: conflicting overloads: public fun perimeter(sideLength: Double): Double
defined in root package in file EquilateralTriangle.kt, public fun perimeter(sideLength: Doubl
e): Double defined in root package in file Square.kt
fun perimeter(sideLength: Double): Double{
^
```

Actually, the problem is that the `perimeter(Double)` function is defined two times. And it would be a mistake to rename our functions to solve the problem, as their names are pretty clear.

To solve this problem, Kotlin allows us to separate source files into packages. So if a function is defined in two different packages, Kotlin will consider them to be distinct functions.

To specify that a source file is part of a package, you just have to add the `package` keyword—followed by the name of the package you would like to set—on the first line of the file.

Furthermore, in order to have a hierarchy of packages, each package can have subpackages. You just need to separate them with a dot. So, `geometry.angles` and `geometry.circles` are two packages (`angles` and `circles`) that are parts of the `geometry` package.

So, the first line of each file becomes:

Geometry.kt

```
package geometry
```

Circle.kt

```
package geometry.circle
```

Angle.kt

```
package geometry.angles
```

Constants.kt

```
package geometry.constants
```

Square.kt

```
package geometry.quadrilateral.square
```

EquilateralTriangle.kt

```
package geometry.triangle.equilateral
```

Now, we just need to compile our program. Let's try.

macOS and Linux :

```
rm -rf ./build/* && kotlinc ./* -d build
```

Windows :

```
del /S/Q build && kotlinc ./* -d build
```

We now get many errors. The problem is that the Kotlin compiler cannot find the `circumference` function or any other reference. Why can't it find them?

If we do not specify which package a file should go in, Kotlin will automatically add our files to the original root package. So, before specifying packages, Kotlin considered all files to be part of the same package. And when calling a function, Kotlin will automatically look for it in the package of the file calling the function.

Now, when calling `printAngleDeg(PI)` in `Geometry.kt`, Kotlin will not try to find it in the root package. Instead, it will look for it in the `geometry` package that we specified for this file. But the `PI` constant is in a different package, so it cannot find the definition of this constant.

So in order for Kotlin to find our `PI` constant, we need to tell it in which package the constant can be found. To do so, we have to write the package name, followed by a dot, followed by `PI`. So `PI` now becomes `geometry.constants.PI`.

Now the code in our files becomes:

Geometry.kt

```
package geometry

/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printAngleDeg(geometry.constants.PI)
}

/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+geometry.circle.circumference(diameter))
}

/**
 * Prints the conversion of an angle measured in radian to
 */
fun printAngleDeg(angRad: Double){
   println("toDegrees("+angRad+")="+geometry.angles.toDegrees(angRad))
}
```

Angles.kt

```
package geometry.angles

/** Flat angle measured in degrees */
const val DEGREES_FLAT_ANGLE = 180
/** Flat angle measured in radians */
const val RADIANS_FLAT_ANGLE = geometry.constants.PI

/**
 * Converts an angle measured in radians to an equivalent angle measured in degrees.
 * @param angrad an angle, in radians
 * @return the measurement of the angle [angrad] in degrees
 */
fun toDegrees(angrad: Double): Double{
   return angrad/RADIANS_FLAT_ANGLE*DEGREES_FLAT_ANGLE
}

/**
 * Converts an angle measured in degrees to an equivalent angle measured in radians.
 * @param angdeg an angle, in degrees
 * @return the measurement of the angle [angdeg] in radians
 */
fun toRadians(angdeg: Double): Double{
   return angdeg/DEGREES_FLAT_ANGLE*RADIANS_FLAT_ANGLE
}
```

Circle.kt

```
package geometry.circle

/**
 * Returns the circumference of a circle when the diameter is [diameter]
 * @param diameter the diameter of the circle
 * @return the circumference of a circle when the diameter is [diameter]
 */
fun circumference(diameter: Double): Double{
   return geometry.constants.PI*diameter
}
```

We can now compile the program (no suspense anymore: it works). Then we can run it:

```
kotlin -cp build GeometryKt
```

This time, an error occurs when we try to run the program:

```
error: could not find or load main class GeometryKt
```

Just like for source files, Kotlin will look for the binary file `GeomtryKt` in the root package when calling `GeometryKt`. But there is now a package assigned to the `Geometry.kt` file. So we just have to tell Kotlin the package in which it can find `GeometryKt`:

```
kotlin -cp build geometry.GeometryKt
```

What we have to do now is edit the `Geometry.kt` file in order to add a call to the two functions we developed:

Geometry.kt

```
package geometry

/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printAngleDeg(geometry.constants.PI)
   printEquilateralTrianglePerimeter(3.0)
   printSquarePerimeter(3.0)
}

/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+geometry.circle.circumference(diameter))
}

/**
 * Prints the conversion of an angle measured in radian to
 */
fun printAngleDeg(angRad: Double) {
   println("toDegrees(" + angRad + ")=" + geometry.angles.toDegrees(angRad))
}

/**
 * Prints the perimeter of an equilateral triangle when the length of a side is [sideLength]
 * @param sideLength the length of a side of the equilateral triangle
 */
fun printEquilateralTrianglePerimeter(sideLength: Double){
   println("Equilateral triangle perimeter(" + sideLength + ")=" + geometry.triangle.equilater
al.perimeter(sideLength))
}

/**
 * Prints the perimeter of a square when the length of a side is [sideLength]
 * @param sideLength the length of a side of the square
 */
fun printSquarePerimeter(sideLength: Double){
   println("Square perimeter(" + sideLength + ")=" + geometry.quadrilateral.square.perimeter(s
ideLength))
}
```

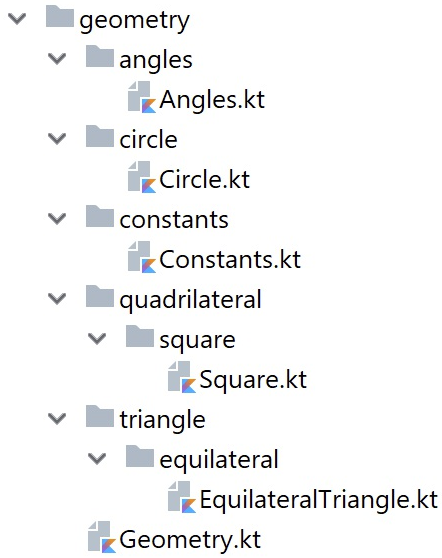
File Tree Convention

With all files in the same directory, it is difficult to know which file is part of which package at a glance.

Furthermore, it is easy to compare the hierarchy of a package using folders constructed to reflect the same hierarchy.

So the approach used most often by developers is to organize source files of a project so each package is associated with a directory, and each source file is in the directory associated with its package.

For our current project, we can use the following file tree:



The command remains the same to compile the project:

macOS and Linux :

```
rm -rf ./build/* && kotlinc ./* -d build
```

Windows :

```
del /S/Q build && kotlinc ./* -d build
```

Imports

Just like in the `Geometry.kt` file, in order to calculate circumference of a circle, we need to call the `geometry.circle.circumference()` function.

As we saw in the second chapter, it is better to keep the code as simple and compact as possible. Imagine if we had to call `geometry.circle.circumference()` each time: it is neither simple nor compact.

In order to address this issue, Kotlin allows us to use imports. An import allows us to specify the complete path to a function in a file. Here is an example using `Geometry.kt` :

```
package geometry

import geometry.circle.circumference


// ...
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+circumference(diameter))
}
// ...
```

The `//...` notation here lets you know that there is some code here that hasn't been printed. This allows you to focus only on a part of the code. Here, only what has changed in Geometry.kt has been displayed; since everything else remained the same, it has been simplified with the `//...` notation.

So we can easily see here how imports make things easier.

Star

Now, let's look at the example of angles and two functions associated with each one. We could write the following import statements:

```
import geometry.angles.toDegrees
import geometry.angles.toRadians
```

But Kotlin provides a way to import everything in a package: the `*` notation. So in order to import all functions in the `geometry.angles` package, we could use:

```
import geometry.angles.*
```

So both functions can now be called in their short form in the code.

Although it is possible, we strongly discourage you from using this notation for two main reasons:

- Even if you call everything in a package today, you may add functions to the package tomorrow. This would make the import useless and create a loss in terms of performance.
- It is difficult to keep track of what is imported and why, which makes the code more complex to read.

So, in most cases, it is better to use full imports and avoid ones that use the star notation. However, it is still important to know what the star notation stands for when you encounter it in code.

Limitations

It is not possible to import two packages containing functions with the same name. So it is not possible to use the following imports:

```
import geometry.quadrilateral.square.perimeter
import geometry.triangle.equilateral.perimeter
```

The reason is the same as the one that motivates us to use packages. If we use both imports, the compiler will not know which `perimeter` function to use when calling it.

Exercises

Exercise 1 - Easy

Simplify all function calls to make them appear in their short forms. As discussed earlier, only `perimeter` functions should not be simplified.

Exercise 2 - Medium

If you understood this chapter well, it will be possible for you to simplify one call to a `perimeter` function. So simplify one call to this function.