The background features a blue and white geometric pattern. A large blue triangle is positioned in the upper right, while a smaller blue triangle is in the upper left. A white diagonal band runs from the top-left corner towards the bottom-right.

Marcin Moskała

Kotlin Essentials



Kotlin Essentials

Marcin Moskała

This book is for sale at
http://leanpub.com/kotlin_developers

This version was published on 2022-12-15

ISBN 978-83-966847-0-7



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 Marcin Moskała

Подписывайся на топовый канал по Kotlin:
<https://t.me/KotlinSenior>

Contents

Introduction	1
For whom is this book written?	1
What will be covered?	1
The Kotlin for Developers series	2
My story	3
Conventions	3
Code conventions	4
Acknowledgments	5
What is Kotlin?	7
Kotlin platforms	7
The Kotlin IDE	11
Where do we use Kotlin?	11
Your first program in Kotlin	13
Live templates	15
What is under the hood on JVM?	16
Summary	19
Variables	20
Basic types, their literals, and operations	24
Numbers	24
Booleans	35
Characters	37
Strings	38
Summary	41
Conditional statements: if, when, try, and while	43
if-statement	43
when-statement	49
when-statement with a value	51
is check	53
Explicit casting	54

CONTENTS

Smart-casting	56
While and do-while statements	58
Summary	60
Functions	61
Single-expression functions	64
Functions on all levels	65
Parameters and arguments	67
Unit return type	68
Vararg parameters	69
Named parameter syntax and default arguments	71
Function overloading	73
Infix syntax	74
Function formatting	76
Summary	80
The power of the for-loop	81
Ranges	83
Break and continue	85
Use cases	86
Summary	90
Nullability	91
Safe calls	92
Not-null assertion	94
Smart-casting	95
The Elvis operator	96
Extensions on nullable types	98
null is our friend	100
lateinit	101
Summary	103
Classes	104
Member functions	104
Properties	106
Constructors	113
Classes representing data in Kotlin and Java	117
Inner classes	118
Summary	120
Inheritance	121
Overriding elements	122
Parents with non-empty constructors	123
Super call	124

CONTENTS

Abstract class	125
Interfaces	128
Visibility	134
Any	140
Summary	140
Data classes	142
Transforming to a string	145
Objects equality	147
Hash code	148
Copying objects	149
Destructuring	152
When and how should we use destructuring?	153
Data class limitations	155
Prefer data classes instead of tuples	156
Summary	160
Objects	162
Object expressions	162
Object declaration	166
Companion objects	167
Data object declarations	174
Constant values	175
Summary	176
Exceptions	178
Throwing exceptions	179
Defining exceptions	180
Catching exceptions	181
Try-catch block used as an expression	182
Finally block	184
Important exceptions	185
Hierarchy of exceptions	187
Summary	188
Enum classes	189
Data in enum values	192
Enum classes with custom methods	193
Summary	194
Sealed classes and interfaces	195
Sealed classes and <code>when</code> expressions	197
Sealed vs enum	199
Use-cases	199

CONTENTS

Summary	201
Annotation classes	202
Meta-annotations	204
Annotating primary constructor	205
List literals	205
Summary	206
Extensions	207
Extension functions under the hood	209
Extension properties	211
Extensions vs members	212
Extension functions on object declarations	215
Member extension functions	216
Use-cases	217
Summary	222
Collections	223
The hierarchy of interfaces	224
Mutable vs read-only types	225
Creating collections	227
Lists	229
Sets	235
Maps	240
Using arrays in practice	246
Summary	253
Operator overloading	254
An example of operator overloading	255
Arithmetic operators	256
The rangeUntil operator	257
The in operator	258
The iterator operator	259
The equality and inequality operators	261
Comparison operators	262
The indexed access operator	264
Augmented assignments	265
Unary prefix operators	267
Increment and decrement	268
The invoke operator	269
Precedence	270
Summary	272
The beauty of Kotlin's type system	273

CONTENTS

What is a type?	273
Why do we have types?	275
The relation between classes and types	277
Class vs type in practice	277
The relationship between types	279
The subtype of all the types: Nothing	282
The result type from return and throw	284
When is some code not reachable?	287
The type of null	289
Summary	291
Generics	292
Generic functions	293
Generic classes	295
Generic classes and nullability	297
Generic interfaces	298
Type parameters and inheritance	301
Type erasure	303
Generic constraints	304
Star projection	306
Summary	308
Final words	309

Introduction

Kotlin is a powerful language, largely thanks to its expressive syntax, intuitive and null-safe type system, and great tooling support. It's no wonder Kotlin is the most popular language for Android development and is a popular alternative to Java for backend applications. It is also used for data science and for multiplatform, iOS, desktop, and web application development. In this book, you are going to learn about the most important Kotlin features, which will let you properly start your adventure with Kotlin.

For whom is this book written?

This book is dedicated to developers. I assume that all developers understand what functions, if-statements, or strings are. However, I try to explain (at least briefly) all the topics that might not be clear for all kinds of developers, like classes, enums, or lists. So I assume this book might be read by C, JavaScript, or Matlab developers.

Kotlin is a multiplatform language, but it is mainly used on JVM. Also, most Kotlin developers have experience in Java. This is why I sometimes reference Java and its platform, and I sometimes present JVM-specific elements. Whenever I do this, I state it explicitly. I assume that some readers might be mainly interested in using Kotlin/JS or Kotlin/Native, so everything that is not described as Java-specific should be useful for them.

What will be covered?

In this book, I cover the topics that I find essential for programming in Kotlin, including:

- variables, values, and types,
- conditional statements and loops,

- support for nullability,
- classes, interfaces, and inheritance,
- object expressions and declarations,
- data, sealed, enum, and annotation classes,
- exceptions,
- extension functions,
- collections,
- operator overloading,
- the type system,
- generics.

This book does not cover functional Kotlin features, like lambda expressions or function types. All the functional features are covered in the continuation of this book: *Functional Kotlin*.

The Kotlin for Developers series

This book is the first in a series of books called *Kotlin for Developers*, which includes the following books:

- *Kotlin Essentials*. This covers all the basic Kotlin features.
- *Functional Kotlin*. This is dedicated to functional Kotlin features, including function types, lambda expressions, collection processing, DSLs, and scope functions.
- *Advanced Kotlin*. This is dedicated to advanced Kotlin features, including generic variance modifiers, delegation, documenting code, string processing, reflection, and multiplatform programming.

This book and my two other books, *Kotlin Coroutines* and *Effective Kotlin*, make a big series covering everything that I believe is needed to become an amazing Kotlin developer.

My story

My story with Kotlin started in 2015 when I worked as a Java Android developer. I was quite frustrated with all the boilerplate code, like getters and setters for every field, and the nearly identical `equals`, `toString`, and `hashCode` methods that are often repeated in many classes. Then, I found the Kotlin preview on the JetBrains website, and I became so fascinated that I dedicated every free moment to exploring this language. Soon after that, I got a job in Kotlin development and became immersed in the community. Now, I have been professionally using Kotlin for over seven years, during which I have published hundreds of articles about Kotlin and a bunch of books, and I have conducted over a hundred workshops about Kotlin. I have become an official JetBrains partner for teaching Kotlin, and a Google Developer Expert in Kotlin. I accumulated a lot of knowledge during all these experiences, so I decided to express it in the form of a series of books, which I call *Kotlin for Developers*.

Conventions

When I mean a concrete element from code, I will use code font. To name a concept, I will use uppercase. To reference an arbitrary element of some type, I will use lowercase. For example:

- `List` is a type or an interface, so it is printed in code font (as in “Function needs to return `List`”),
- `List` represents a concept, so it starts with uppercase (as in “This explains the essential difference between `List` and `Set`”),
- a `list` is an instance, which is why it is lowercase (as in “the `list` variable holds a `list`”).

In this book, I decided to use a dash between “if”, “when”, “try”, “while”, and “for”, and the word describing it, like “condition”, “loop”, “statement”, or “expression”. I do this to

improve readability. So, I will write “if-condition” instead of “if condition”, or “while-loop” instead of “while loop”. “if” and “when” are conditions, “while” and “for” are loops. All of them can be used as statements, while “if”, “when” and “try” can also be used as expressions. I also decided not to use a dash after “if-else”, “if-else-if”, or “try-catch” and their descriptor, like in an “if-else statement”.

Code conventions

Most of the presented snippets are executable code with no import statements. In the online version of this book on the Kt. Academy website, most snippets can be executed so that readers can play with the code.

Snippet results are presented using the `println` function. The result will often be placed after the statement. Here is an example:

```
fun main() {
    println("Hello") // Hello
    println(10 + 20) // 30
}
```

Acknowledgments



Owen Griffiths has been developing software since the mid 1990s and remembers the productivity of languages such as Clipper and Borland Delphi. Since 2001, he has moved to web, server-based Java, and the open-source revolution. With many years of commercial Java experience, he picked up Kotlin in early 2015. After taking detours into Clojure and Scala, like Goldilocks, he thinks Kotlin is just right and tastes the best. Owen enthusiastically helps Kotlin developers continue to succeed.



Nicola Corti is a Google Developer Expert for Kotlin. He has been working with this language since before version 1.0 and is the maintainer of several open-source libraries and tools for mobile developers (Detekt, Chucker, AppIntro). He's currently working in the React Native core team at Meta, building one of the most popular cross-platform mobile frameworks. Furthermore, he is an active member of the developer community. His involvement goes from speaking at international conferences to being a member of CFP committees and supporting developer communities across Europe. In his free time, he also loves baking, podcasting, and running.



Matthias Schenk started his career with Java over ten years ago, mainly in the Spring/Spring Boot Ecosystem. Eighteen months ago, he switched to Kotlin and has since become a big fan of working with native Kotlin frameworks like Koin, Ktor, and Exposed.



created.

Endre Deak is a software architect building AI infrastructure at Disco, a market-leading legal tech company. He has 15 years of experience building complex scalable systems, and he thinks Kotlin is one of the best programming languages ever



Emanuele Papa is passionate about Android and has been fascinated by it since 2010: the more he learns, the more he wants to share his knowledge with others, which is why he started his own blog. In his current role as Senior Android Developer at Zest One, he is now focusing on

Kotlin Multiplatform Mobile: he has already given a few talks on this topic.

Roman Kamyshnikov, PhD in engineering, is an Android developer who started his career with Java but switched to Kotlin at the beginning of 2020. His professional interests include architecture patterns, TDD, functional programming, and Jetpack Compose. Author of several articles about Android and Kotlin Coroutines.

Grigory Pletnev has been a software engineer since 2000, mostly in the embedded domain. He joined the Android developer community in 2010. Having got acquainted with Kotlin in 2017, he began to use it in pet projects, gradually migrating Harman Connected Services and its customers' projects to Kotlin. He also has a passion for languages, sailing, and mead making.

What is Kotlin?

Kotlin is an open-source, multiplatform, multi-paradigm, statically typed, general-purpose programming language. But what does all this mean?

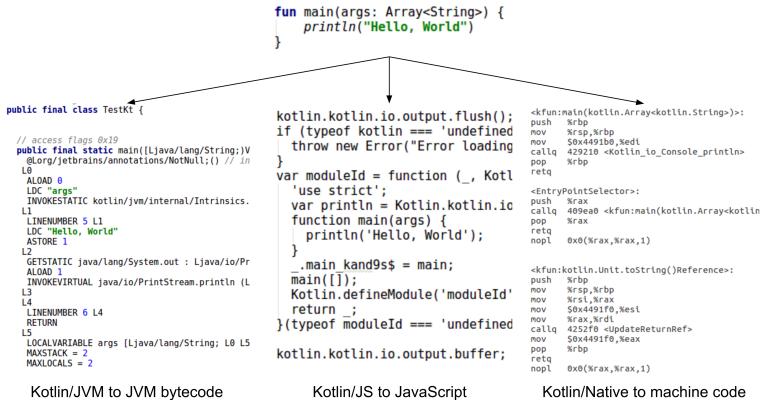
- Open-source means that the sources of the Kotlin compiler are freely available for modification and redistribution. Kotlin is primarily made by JetBrains, but now there is the Kotlin Foundation, which promotes and advances the development of this language. There is also a public process known as KEEP, which allows anyone to see and comment on official design change propositions.
- Multiplatform means that a language can be used on more than one platform. For instance, Kotlin can be used both on Android and iOS.
- Multi-paradigm means that a language has support for more than one programming paradigm. Kotlin has powerful support for both Object-Oriented Programming and Functional Programming.
- Statically typed means that each variable, object, and function has an associated type that is known at compile time.
- General-purpose means that a language is designed to be used for building software in a wide variety of application domains across a multitude of hardware configurations and operating systems.

These descriptions might not be clear now, but you will see them all in action throughout the book. Let's start by discussing Kotlin's multiplatform capabilities.

Kotlin platforms

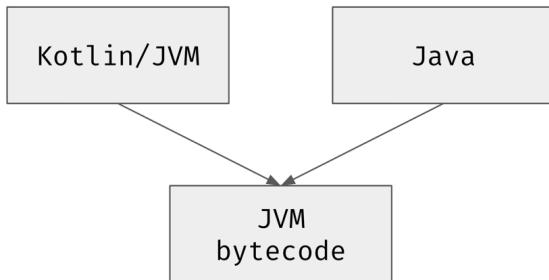
Kotlin is a compiled programming language. This means that you can write some code in Kotlin and then use the Kotlin

compiler to produce code in a lower-level language. Kotlin can currently be compiled into JVM bytecode (Kotlin/JVM), JavaScript (Kotlin/JS), or machine code (Kotlin/Native).

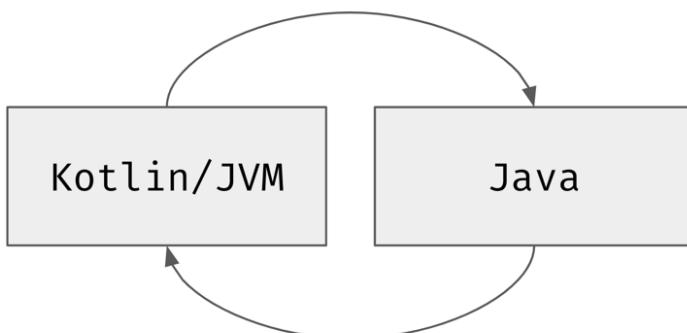


In this book, I would like to address all these compilation targets and, by default, show code that works on all of them, but I will concentrate on the most popular one: Kotlin/JVM.

Kotlin/JVM is the technology that's used to compile Kotlin code into JVM bytecode. The result is nearly identical to the result of compiling Java code into JVM bytecode. We also use the term "Kotlin/JVM" to talk about code that will be compiled into JVM bytecode.

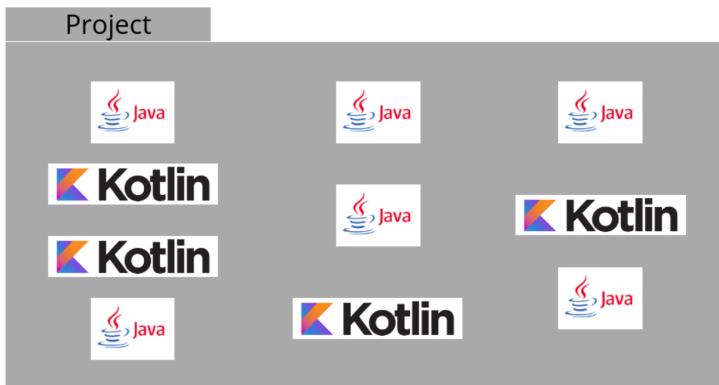


Kotlin/JVM and Java are fully interoperable. Any code written in Java can be used in Kotlin/JVM. Any Java library, including those based on annotation processing, can be used in Kotlin/JVM. Kotlin/JVM can use Java classes, modules, libraries, and the Java standard library. Any Kotlin/JVM code can be used in Java (except for suspending functions, which are a support for Kotlin Coroutines).



Kotlin and Java can be mixed in a single project. A typical scenario is that a project was initially developed in Java, but then its creators decided to start using Kotlin. To do this, instead of migrating the whole project, these developers decided to

add Kotlin to it. So, whenever they add a new file, it will be a Kotlin file; furthermore, when they refactor old Java code, they will migrate it to Kotlin. Over time, there is more and more Kotlin code until it excludes Java completely.



One example of such a project is the Kotlin compiler itself. It was initially written in Java, but more and more files were migrated to Kotlin when it became stable enough. This process has been happening for years now; at the time of writing this book, the Kotlin compiler project still contains around 10% of Java code.

Now that we understand the relationship between Kotlin and Java, it is time to fight some misconceptions. Many see Kotlin as a layer of syntactic sugar on top of Java, but this is not true. Kotlin is a different language than Java. It has its own conventions and practices, and it has features that Java does not have, like multiplatform capabilities and coroutines. You don't need to know Java to understand Kotlin. In my opinion, Kotlin is a better first language than Java. Junior Kotlin developers do not need to know what the `equals` method is and how to override it. For them, it is enough to know the default class and data class equality¹. They don't need to learn to write getters and setters, or how to implement a singleton or a builder pattern. Kotlin has a lower entry threshold than

¹It will be explained in the chapter Data classes.

Java and does not need the JVM platform.

The Kotlin IDE

The most popular Kotlin IDEs (integrated development environments) are IntelliJ IDEA and Android Studio. However, you can also write programs in Kotlin using VS Code, Eclipse, Vim, Emacs, Sublime Text, and many more. You can also write Kotlin code online, for instance, using the official online IDE that can be found at this link play.kotlinlang.org.

Where do we use Kotlin?

Kotlin can be used as an alternative to Java, JavaScript, C++, Objective-C, etc. However, it is most mature on JVM, so it is currently mainly used as an alternative to Java.

Kotlin has become quite popular for backend development. I most often see it used with the Spring framework, but some projects use Kotlin with backend frameworks like Vert.x, Ktor, Micronaut, http4k or Javalin.

Kotlin has also practically become the standard language for Android development. Google has officially suggested that all Android applications should be written in Kotlin² and has announced that all their APIs will be designed primarily for Kotlin³.

More and more projects are now taking advantage of the fact that Kotlin can be compiled for a few different platforms because this means that teams can write code that runs on both Android and iOS, or on both the backend and the frontend. Moreover, this cross-platform compatibility means that library creators can create one library for multiple platforms at the same time. Kotlin's multiplatform capabilities are already

²Source: <https://techcrunch.com/2022/08/18/five-years-later-google-is-still-all-in-on-kotlin/>

³Source: <https://developer.android.com/kotlin/first>

being used in many companies, and they are getting more and more popular.

It is also worth mentioning Jetpack Compose, which is a toolkit for building native UIs in Kotlin. It was initially developed for Android, but it uses Kotlin's multiplatform capabilities and can also be used to create views for websites, desktop applications, iOS applications, and other targets⁴.

A lot of developers are using Kotlin for front-end development, mainly using React, and there is also a growing community of data scientists using Kotlin.

As you can see, there is already a lot that you can do in Kotlin, and there are more and more possibilities as each year passes. I am sure you will find good ways to apply your new knowledge once you've finished reading this book.

⁴At the moment, the maturity of these targets differs.

Подписывайся на топовый канал по Kotlin:
<https://t.me/KotlinSenior>

Your first program in Kotlin

The first step in our Kotlin adventure is to write a minimal program in this language. Yes, it's the famous “Hello, World!” program. This is what it looks like in Kotlin:

```
fun main() {  
    println("Hello, World")  
}
```

This is minimal, isn't it? We need no classes (like we do in Java), no objects (like `console` in JavaScript), and no conditions (like in Python when we start code in the IDE). We need the `main` function and the `println` function call with some text⁵.

This is the most popular (but not the only) variant of the “main” function. If we need arguments, we might include a parameter of type `Array<String>`:

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

There are also other forms of the `main` function:

```
fun main(vararg args: String) {  
    println("Hello, World")  
}
```

⁵The `println` function is implicitly imported from the standard library package `kotlin.io`.

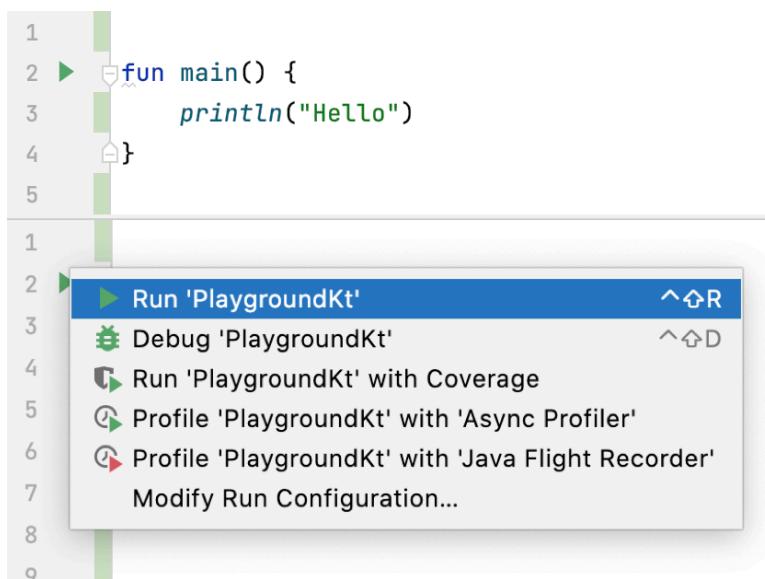
```
class Test {  
    companion object {  
        @JvmStatic  
        fun main(args: Array<String>) {  
        }  
    }  
}  
  
  
suspend fun main() {  
    println("Hello, World")  
}
```

Although these are all valid, let's concentrate on the simple `main` function as we will find it most useful. I will use it in nearly every example in this book. Such examples are usually completely executable if you just copy-paste them into IntelliJ or the Online Kotlin Playgroun⁶.

```
fun main() {  
    println("Hello, World")  
}
```

All you need to do to start the `main` function in IntelliJ is click the green triangle which appears on the left side of the `main` function; this is called the “gutter icon”, also known as the “Run” button.

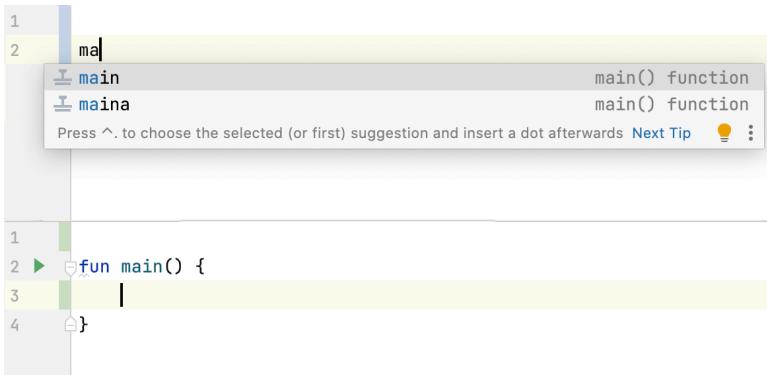
⁶You can also find some chapters of this book online on the Kt. Academy website. These examples can be started and modified thanks to the Kotlin Playgroun feature.



Live templates

If you decide to test or practice the material from this book⁷, you will likely be writing the `main` function quite often. Here come live templates to help us. This is an IntelliJ feature that suggests using a template when you start typing its name in a valid context. So, if you start typing “main” or “maina” (for `main` with arguments) in a Kotlin file, you will be shown a suggestion that offers the whole `main` function.

⁷It makes me happy when people try to challenge what I am teaching. Be skeptical, and verify what you’ve learned; this is a great way to learn something new and deepen your understanding.



In most my workshops, I've used this template hundreds of times. Whenever I want to show something new with live coding, I open a “Playground” file, select all its content (Ctrl/Command + A), type “main”, confirm the live template with Enter, and I have a perfect space for showing how Kotlin works.

I also recommend you test this now. Open any Kotlin project (it is best if you have a dedicated project for playing with Kotlin), create a new file (you can name it “Test” or “Playground”), and create the `main` function with the live template “maina”. Use the `print` function with some text, and run the code with the Run button.

What is under the hood on JVM?

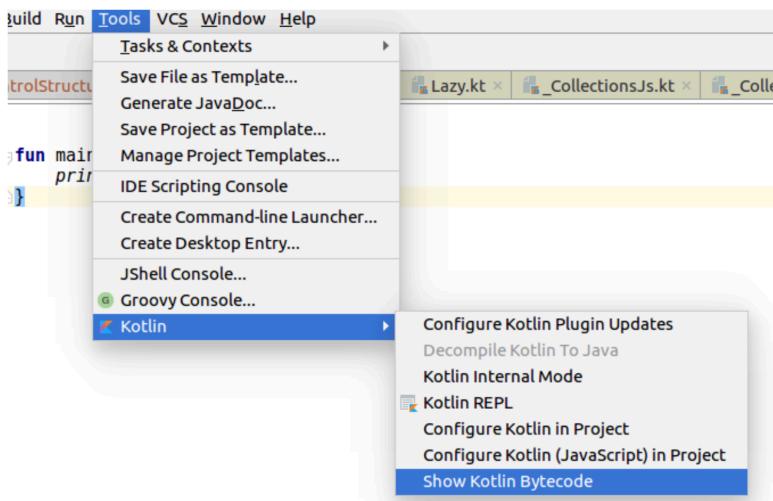
The most important target for Kotlin is JVM (Java Virtual Machine). On JVM, every element needs to be in a class. So, you might be wondering how it is possible that our main function can be started there if it is not in a class. Let's figure it out. On the way, we will learn to find out what our Kotlin code would look like if it were written in Java. This is Java developers' most powerful tool for learning how Kotlin works.

Let's start by opening or starting a Kotlin project in IntelliJ or Android Studio. Make a new Kotlin file called “Playground”. Inside this, use the live template “maina” to create the main

function with arguments and add `println("Hello, World")` inside.

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

Now, select from the tabs Tools > Kotlin > Show Kotlin Bytecode.



On the right side, a new tool should open. “Show Kotlin Bytecode” shows the JVM bytecode generated from this file.

The screenshot shows the IntelliJ IDEA interface with the code editor and the decompiler tool window. The code editor contains the following Kotlin code:

```

1 fun main(args: Array<String>) {
2     println("Hello, World")
3 }

```

The decompiler tool window shows the corresponding JVM bytecode. The bytecode is as follows:

```

1 // ======PlaygroundKt.class ======
2 // class version 52.0 (52)
3 // access flags 0x11
4 public final class PlaygroundKt {
5
6
7     // access flags 0x19
8     public final static main([Ljava/lang/String;)V
9         // annotable parameter count: 1 (visible)
10        // annotable parameter count: 1 (invisible)
11        // @org/jetbrains/annotations/NotNull();() // invisible
12        L0
13            ALOAD 0
14            LDC "args"
15            INVOKESTATIC kotlin/jvm/internal/Intrinsics.checkNotNull
16            L1
17            LINENUMBER 3 L1
18            LDC "Hello, World"
19            ASTORE 1
20            L2
21            GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
22            ALOAD 1
23            INVOKEVIRTUAL java/io/PrintStream.println (Ljava/la

```

This is a great place for everyone who likes reading JVM bytecode. Since not everyone is Jake Wharton, most of us might find the “Decompile” button useful. What it does is quite funny. We’ve just compiled our Kotlin code into JVM bytecode, and this button decompiles this bytecode into Java. As a result, we can see what our code would look like if it were written in Java⁸.

```

public final class PlaygroundKt {
    public static final void main(@NotNull String[] args) {
        Intrinsics.checkNotNullParameter(args, paramName: "args");
        String var1 = "Hello, World";
        System.out.println(var1);
    }
}

```

This code reveals that our `main` function on JVM becomes a static function inside a class named `PlaygroundKt`. Where does this name come from? Try to guess. Yes, this is, by default, the file’s name with the “Kt” suffix. The same happens to all other functions and properties defined outside of classes on JVM.

The name of `PlaygroundKt` can be changed by adding the `@file:JvmName("NewName")` annotation at the top of the file⁹. However, this does not change how elements defined in this

⁸This doesn’t always work because the decompiler is not perfect, but it is really helpful anyway.

⁹More about this in the book *Advanced Kotlin*, chapter *Kotlin and Java interoperability*.

file are used in Kotlin. It only influences how we will use such functions from Java. If we wanted to call our `main` function from Java code, we can call `PlaygroundKt.main({})`.

If you have experience with Java, remember this tool as it can help you to understand:

- How Kotlin code works on a low level.
- How a certain Kotlin feature works under the hood.
- How to use a Kotlin element in Java.

There are proposals to make a similar tool to show JavaScript generated from Kotlin code when our target is Kotlin/JS. However, at the time of writing this book, the best you can do is to open the generated files yourself.

Summary

We've learned about using `main` functions and creating them easily with live templates. We've also learned how to find out what our Kotlin code would look like if it were written in Java. For me, it seems like we have quite a nice toolbox for starting our adventure. So, without further ado, let's get on with that.

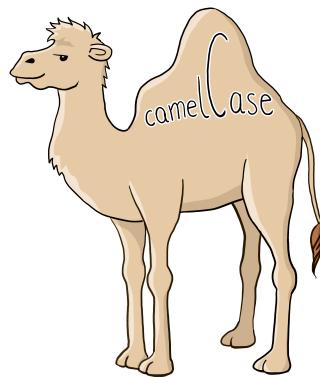
Variables

To declare a variable in Kotlin, we start with the `val` or `var` keyword, then a variable name, the equality sign, and an initial value.

- The keyword `var` (which stands for “variable”) represents *read-write variables* and is used to define variables whose values can be reassigned after initialization. This means that if you use `var`, you can always assign a new value to this variable.
- The keyword `val` (which stands for “value”) represents *read-only variables* and is used to define values that cannot be reassigned. This means that if you use `val`, you cannot assign a new value to this variable once it is initialized.

```
fun main() {  
    val a = 10  
    var b = "ABC"  
    println(a) // 10  
    println(b) // ABC  
    // a = 12 is not possible, because a is read-only!  
    b = "CDE"  
    println(b) // CDE  
}
```

We can name variables using characters, underscore `_`, and numbers (but numbers are not allowed at the first position). By convention, we name variables with the camelCase convention; this means the variable name starts with a lowercase letter, then (instead of using spaces) each next word starts with a capital letter.



In Kotlin, we name variables using camelCase.

Variables don't need to specify their type explicitly, but this doesn't mean that variables are not types. Kotlin is a statically typed language, therefore every variable needs its type specified. The point is that Kotlin is smart enough to infer the type from the value that is set. `10` is of type `Int`, so the type of `a` in the above example is `Int`. "ABC" is of type `String`, so the type of `b` is `String`.

```
fun main() {  
    val a = 10  
    var b = "ABC"  
  
    a  
}
```

The code shows a snippet of Kotlin code. The variable `a` is highlighted with a yellow background and has a tooltip below it showing the value `a` and the type `Int`.

We can also specify a variable type explicitly using a colon and a type **after** the variable name.

```
fun main() {  
    val a: Int = 10  
    var b: String = "ABC"  
    println(a) // 10  
    println(b) // ABC  
    b = "CDE"  
    println(b) // CDE  
}
```

When we initialize a variable, we should give it a value. As in the example below, a variable's definition and initialization can be separated if Kotlin can be sure that the variable won't be used before any value is set. I suggest avoiding this practice when it's not necessary.

```
fun main() {  
    val a: Int  
    a = 10  
    println(a) // 10  
}
```

We can assume that a variable should normally be initialized by using an equality sign after its declaration (like in `val a = 10`). So, what can stand on the right side of the assignment? It can be any expression so a piece of code that returns a value. Here are the most important types of expressions in Kotlin:

- a basic type literal, like `1` or `"ABC"`¹⁰,
- a conditional statement used as an expression, like `if-expression`, `when-expression`, or `try-catch expression`¹¹.
- a constructor call¹²,
- a function call¹³,

¹⁰Basic types are covered in the next chapter.

¹¹Conditional statements are covered in the chapter [Conditional statements](#).

¹²Constructors are covered in the chapter [Classes](#)

¹³All functions in Kotlin declare a result type, so they can all be used on the right side of a variable assignment. Functions are covered in the chapter [Functions](#).

- an object expression or an object declaration¹⁴,
- a function literal, like a lambda expression, an anonymous function, or a function reference¹⁵,
- an element reference¹⁶.

We have a lot to discuss, so let's start with basic type literals.

¹⁴Object expression and object declaration are covered in the chapter Objects

¹⁵All the function literal types are covered in the book Functional Kotlin in chapters *Anonymous functions*, *Lambda expressions*, and *Function references*.

¹⁶All the different kinds of element references are covered in the book Advanced Kotlin, in chapter Reflection.



Basic types, their literals, and operations

Every language needs a convenient way to represent basic kinds of values, like numbers or characters. All languages need to have built-in **types** and **literals**. Types are used to represent certain types of values. Some type examples are `Int`, `Boolean`, or `String`. Literals are built-in notations that are used to create instances. Some literal examples are a string literal, which is text in quotation marks, or an integer literal, which is a bare number.

In this chapter, we'll learn about the basic Kotlin types and their literals:

- numbers (`Int`, `Long`, `Double`, `Float`, `Short`, `Byte`),
- booleans (`Boolean`),
- characters (`Char`),
- strings (`String`).

There is also the array primitive type in Kotlin, which will be covered in the chapter Collections.

In Kotlin, all values are considered objects (there are no primitive types), so they all have methods, and their types can be used as generic type arguments (this will be covered later). Types that represent numbers, booleans, and characters might be optimized by the Kotlin compiler and used as primitives, but this optimization does not affect Kotlin developers, therefore you don't need to even think about it.

Let's start discussing the basic types in Kotlin, one by one.

Numbers

In Kotlin, there is a range of different types that are used to represent numbers. They can be divided into those representing integer numbers (without decimal points) and those

representing floating-point numbers (with decimal points). In these groups, the difference is in the number of bits used to represent these numbers, which determines the possible number size and precision.

To represent integer numbers, we use `Int`, `Long`, `Byte`, and `Short`.

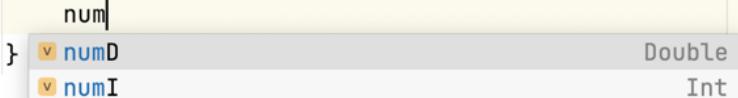
Type	Size (bits)	Min value	Max value
<code>Byte</code>	8	-128	127
<code>Short</code>	16	-32768	32767
<code>Int</code>	32	-2^{31}	$2^{31} - 1$
<code>Long</code>	64	-2^{63}	$2^{63} - 1$

To represent floating-point numbers, we use `Float` and `Double`.

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
<code>Float</code>	32	24	8	6-7
<code>Double</code>	64	53	11	15-16

A plain number without a decimal point is interpreted as an `Int`. A plain number with a decimal point is interpreted as a `Double`.

```
fun main() {
    val numI = 42
    val numD = 3.14
    num|
```



You can create `Long` by using the `L` suffix after the number. `Long` is also used for number literals that are too big for `Int`.

```
fun main() {  
    val numL1 = 42L  
    val numL2 = 12345678912345  
    num|  
} v numL1 Long  
v numL2 Long
```

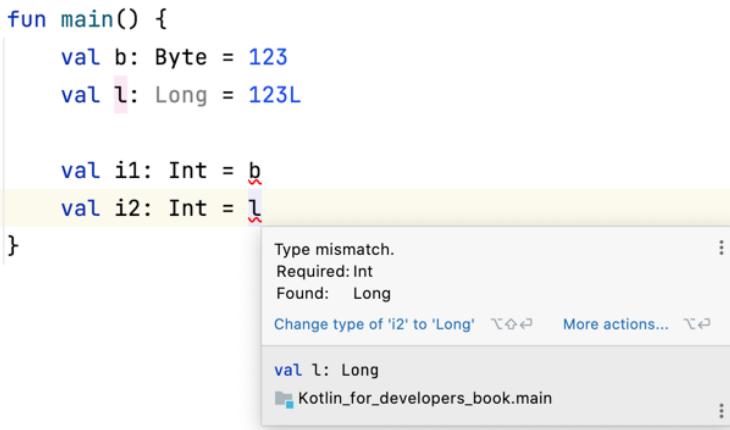
Similarly, you can create a `Float` by ending a number with the `F` or `f` suffix.

```
fun main() {  
    val numF1 = 42F  
    val numF2 = 123.45F  
    num|  
} v numF1 Float  
v numF2 Float
```

There is no suffix to create `Byte` or `Short` types. However, a number explicitly typed as one of these types will create an instance of this type. This also works for `Long`.

```
fun main() {  
    val b: Byte = 123  
    val s: Short = 345  
    val l: Long = 345  
}
```

This is not a conversion! Kotlin does not support implicit type conversion, so you cannot use `Byte` or `Long` where `Int` is expected.



If we need to explicitly convert one number to another type, we use explicit conversion functions like `toInt` or `toLong`.

```
fun main() {  
    val b: Byte = 123  
    val l: Long = 123L  
    val i: Int = 123  
  
    val i1: Int = b.toInt()  
    val i2: Int = l.toInt()  
    val l1: Long = b.toLong()  
    val l2: Long = i.toLong()  
}
```

Underscores in numbers

In number literals, we can use the underscore `_` between digits. This character is ignored, but we sometimes use it to format long numbers for better readability.

```
fun main() {
    val million = 1_000_000
    println(million) // 1000000
}
```

Other numeral systems

To define a number using the hexadecimal numeral system, start it with `0x`. To define a number using the binary numeral system, start it with `0b`. The octal numeral system is not supported.

```
fun main() {
    val hexBytes = 0xA4_D6_FE_FE
    println(hexBytes) // 2765553406
    val bytes = 0b01010010_01101101_11101000_10010010
    println(bytes) // 1382934674
}
```

Number and conversion functions

All basic types that represent numbers are a subtype of the `Number` type.

```
fun main() {
    val i: Int = 123
    val b: Byte = 123
    val l: Long = 123L

    val n1: Number = i
    val n2: Number = b
    val n3: Number = l
}
```

The `Number` type specifies transformation functions: from the current number to any other basic type representing a number.

```
abstract class Number {  
    abstract fun toDouble(): Double  
    abstract fun toFloat(): Float  
    abstract fun toLong(): Long  
    abstract fun toInt(): Int  
    abstract fun toChar(): Char  
    abstract fun toShort(): Short  
    abstract fun toByte(): Byte  
}
```

This means that for each basic number you can transform it into a different basic number using the `to{new type}` function. Such functions are known as **conversion functions**.

```
fun main() {  
    val b: Byte = 123  
    val l: Long = b.toLong()  
    val f: Float = l.toFloat()  
    val i: Int = f.toInt()  
    val d: Double = i.toDouble()  
    println(d) // 123.0  
}
```

Operations on numbers

Numbers in Kotlin support the basic mathematical operations:

- addition (+),
- subtraction (-),
- multiplication (*),
- division (/).

```
fun main() {  
    val i1 = 12  
    val i2 = 34  
    println(i1 + i2) // 46  
    println(i1 - i2) // -22  
    println(i1 * i2) // 408  
    println(i1 / i2) // 0  
  
    val d1 = 1.4  
    val d2 = 2.5  
    println(d1 + d2) // 3.9  
    println(d1 - d2) // -1.1  
    println(d1 * d2) // 3.5  
    println(d1 / d2) // 0.5599999999999999  
}
```

Notice, that the correct result of `1.4 / 2.5` should be `0.56`, not `0.5599999999999999`. This problem will be addressed soon.

Beware that when we divide an `Int` by an `Int`, the result is also `Int`, so the decimal part is lost.

```
fun main() {  
    println(5 / 2) // 2, not 2.5  
}
```

The solution is first to convert an integer into a floating-point representation and then divide it.

```
fun main() {  
    println(5.toDouble() / 2) // 2.5  
}
```

There is also a remainder operator¹⁷ %:

```
fun main() {  
    println(1 % 3) // 1  
    println(2 % 3) // 2  
    println(3 % 3) // 0  
    println(4 % 3) // 1  
    println(5 % 3) // 2  
    println(6 % 3) // 0  
    println(7 % 3) // 1  
    println(0 % 3) // 0  
    println(-1 % 3) // -1  
    println(-2 % 3) // -2  
    println(-3 % 3) // 0  
}
```

Kotlin also supports operations that modify a read-write variable var:

- +=, where $a += b$ is the equivalent of $a = a + b$,
- -=, where $a -= b$ is the equivalent of $a = a - b$,
- *=, where $a *= b$ is the equivalent of $a = a * b$,
- /=, where $a /= b$ is the equivalent of $a = a / b$,
- %=, where $a %= b$ is the equivalent of $a = a \% b$,
- post-incrementation and pre-incrementation ++, which increment variables value by 1,
- post-decrementation and pre-decrementation --, which decrement variables value by 1.

¹⁷This operator is similar to modulo. Both the remainder and the modulo operations act the same for positive numbers but differently for negative numbers. The result of -5 remainder 4 is -1 because $-5 = 4 * (-1) + (-1)$. The result of -5 modulo 4 is 3 because $-5 = 4 * (-2) + 3$.

```
fun main() {
    var i = 1
    println(i) // 1
    i += 10
    println(i) // 11
    i -= 5
    println(i) // 6
    i *= 3
    println(i) // 18
    i /= 2
    println(i) // 9
    i %= 4
    println(i) // 1

    // Post-incrementation
    // increments value and returns the previous value
    println(i++) // 1
    println(i) // 2

    // Pre-incrementation
    // increments value and returns the new value
    println(++i) // 3
    println(i) // 3

    // Post-decrementation
    // decrements value and returns the previous value
    println(i--) // 3
    println(i) // 2

    // Pre-decrementation
    // decrements value and returns the new value
    println(--i) // 1
    println(i) // 1
}
```

Operations on bits

Kotlin also supports operations on bits using the following methods, which can be called using the infix notation (so,

between two values):

- `and` keeps only bits that have 1 in the same binary positions in both numbers.
- `or` keeps only bits that have 1 in the same binary positions in one or both numbers.
- `xor` keeps only bits that have exactly one 1 in the same binary positions in both numbers.

```
fun main() {  
    println(0b011 and 0b001) // 1, that is 0b001  
    println(0b011 or 0b001) // 3, that is 0b011  
    println(0b011 xor 0b001) // 2, that is 0b010  
}
```

BigDecimal and BigInteger

All basic types in Kotlin have limited size and precision, which can lead to imprecise or incorrect results in some situations.

```
fun main() {  
    println(0.1 + 0.2) // 0.3000000000000004  
    println(2147483647 + 1) // -2147483648  
}
```

This is a standard tradeoff in programming, and in most cases we just need to accept it. However, there are cases where we need to have perfect precision and unlimited number size. On JVM, for unlimited number size we should use `BigInteger`, which represents a number without a decimal part. For unlimited size and precision, we should use the `BigDecimal`, which represents a number that has a decimal part. Both can be created using constructors¹⁸, factory functions (like `valueOf`), or a conversion from basic types that represent numbers (`toBigDecimal` and `toBigInteger` methods).

¹⁸Constructors will be covered in the chapter Classes.

```
import java.math.BigDecimal
import java.math.BigInteger

fun main() {
    val i = 10
    val l = 10L
    val d = 10.0
    val f = 10.0F

    val bd1: BigDecimal = BigDecimal(123)
    val bd2: BigDecimal = BigDecimal("123.00")
    val bd3: BigDecimal = i.toBigDecimal()
    val bd4: BigDecimal = l.toBigDecimal()
    val bd5: BigDecimal = d.toBigDecimal()
    val bd6: BigDecimal = f.toBigDecimal()
    val bi1: BigInteger = BigInteger.valueOf(123)
    val bi2: BigInteger = BigInteger("123")
    val bi3: BigInteger = i.toBigInteger()
    val bi4: BigInteger = l.toBigInteger()
}
```

BigDecimal and BigInteger also support basic mathematical operators:

```
import java.math.BigDecimal
import java.math.BigInteger

fun main() {
    val bd1 = BigDecimal("1.2")
    val bd2 = BigDecimal("3.4")
    println(bd1 + bd2) // 4.6
    println(bd1 - bd2) // -2.2
    println(bd1 * bd2) // 4.08
    println(bd1 / bd2) // 0.4

    val bil = BigInteger("12")
    val bi2 = BigInteger("34")
    println(bil + bi2) // 46
    println(bil - bi2) // -22
```

```
    println(bi1 * bi2) // 408
    println(bi1 / bi2) // 0
}
```

On platforms other than Kotlin/JVM, external libraries are needed to represent numbers with unlimited size and precision.

Booleans

Another basic type is `Boolean`, which has two possible values: `true` and `false`.

```
fun main() {
    val b1: Boolean = true
    println(b1) // true
    val b2: Boolean = false
    println(b2) // false
}
```

We use booleans to express yes/no answers, like:

- Is the user an admin?
- Has the user accepted the cookies policy?
- Are two numbers identical?

In practice, booleans are often a result of some kind of comparison.

Equality

A `Boolean` is often a result of equality comparison. In Kotlin, we compare two objects for equality using the double equality sign `==`. To check if two objects are not equal, we use the non-equality sign `!=`.

```
fun main() {  
    println(10 == 10) // true  
    println(10 == 11) // false  
    println(10 != 10) // false  
    println(10 != 11) // true  
}
```

Numbers and all objects that are comparable (i.e., they implement the `Comparable` interface) can also be compared with `>`, `<`, `>=`, and `<=`.

```
fun main() {  
    println(10 > 10) // false  
    println(10 > 11) // false  
    println(11 > 10) // true  
  
    println(10 < 10) // false  
    println(10 < 11) // true  
    println(11 < 10) // false  
  
    println(10 >= 10) // true  
    println(10 >= 11) // false  
    println(11 >= 10) // true  
  
    println(10 <= 10) // true  
    println(10 <= 11) // true  
    println(11 <= 10) // false  
}
```

Boolean operations

There are three basic logical operators in Kotlin:

- `and &&`, which returns `true` when the value on both its sides is `true`; otherwise, it returns `false`.
- `or ||`, which returns `true` when the value on either of its sides is `true`; otherwise, it returns `false`.
- `not !`, which turns `true` into `false`, and `false` into `true`.

```
fun main() {  
    println(true && true) // true  
    println(true && false) // false  
    println(false && true) // false  
    println(false && false) // false  
  
    println(true || true) // true  
    println(true || false) // true  
    println(false || true) // true  
    println(false || false) // false  
  
    println(!true) // false  
    println(!false) // true  
}
```

Kotlin does not support any kind of automatic conversion to Boolean (or any other type), so logical operators should be used only with objects of type Boolean.

Characters

To represent a single character, we use the `Char` type. We specify a character using apostrophes.

```
fun main() {  
    println('A') // A  
    println('Z') // Z  
}
```

Each character is represented as a Unicode number. To find out the Unicode of a character, use the `code` property.

```
fun main() {  
    println('A'.code) // 65  
}
```

Kotlin accepts Unicode characters. To describe them by their code, we start with \u, and then we need to use hexadecimal format, just like in Java.

```
fun main() {  
    println('\u00A3') // £  
}
```

Strings

Strings are just sequences of characters that form a piece of text. In Kotlin, we create a string using quotation marks " or triple quotation marks """.

```
fun main() {  
    val text1 = "ABC"  
    println(text1) // ABC  
    val text2 = """DEF"""  
    println(text2) // DEF  
}
```

A string wrapped in single quotation marks requires text in a single line. If we want to define a newline character, we need to use a special character \n. This is not the only thing that needs (or might need) a backslash to be expressed in a string.

Escape Sequence	Meaning
\t	Tab
\b	Backspace
\r	Carriage return
\f	Form feed
\n	Newline
\'	Single quotation mark
\"	Quotation mark
\\\	Backslash

Strings in triple quotation marks can be multiline; in these strings, special characters can be used directly, and forms prefixed by a backslash don't work.

```
fun main() {
    val text1 = "Let's say:\n\"Hooray\""
    println(text1)
    // Let's say:
    // "Hooray"
    val text2 = """Let's say:\n\"Hooray"""
    println(text2)
    // Let's say:\n\"Hooray\"
    val text3 = """Let's say:
    "Hooray"""
    println(text3)
    // Let's say:
    // "Hooray"
}
```

To better format triple quotation mark strings, we use the `trimIndent` function, which ignores a constant number of spaces for each line.

```
fun main() {
    val text = """
        Let's say:
        "Hooray"
    """.trimIndent()
    println(text)
    // Let's say:
    // "Hooray"

    val description = """
        A
        B
        C
    """.trimIndent()
    println(description)
    // A
    // B
    //     C
}
```

String literals may contain template expressions, which are pieces of code that are evaluated and whose results are concatenated into a string. A template expression starts with a dollar sign (\$) and consists of either a variable name (like "text is \$text") or an expression in curly braces (like "1 + 2 = \${1 + 2}").

```
fun main() {
    val name = "Cookie"
    val surname = "DePies"
    val age = 6

    val fullName = "$name $surname ($age)"
    println(fullName) // Cookie DePies (6)

    val fullNameUpper =
        "${name.uppercase()} ${surname.uppercase()} ($age)"
    println(fullNameUpper) // COOKIE DEPIES (6)

    val description = """
        Name: $name
        Surname: $surname
        Age: $age
    """.trimIndent()
    println(description)
    // Name: Cookie
    // Surname: DePies
    // Age: 6
}
```

If you need to use a special character inside a triple quotation mark string, the easiest way is to specify it with a regular string and include it using template syntax.

```
fun main() {
    val text1 = """ABC\nDEF"""
    println(text1) // ABC\nDEF
    val text2 = """ABC${"\n"}DEF"""
    println(text2)
    // ABC
    // DEF
}
```

In Kotlin strings, we use Unicode, so we can also define a Unicode character using a number that starts with \u, and then specifying a Unicode character code in hexadecimal syntax.

```
fun main() {
    println("😊") // 😊
    println("\uD83D\uDC4B") // 🎉
    println("\uD83C\uDDF5\uD83C\uDDF1") // 🎉
```

Summary

In this chapter, we've learned about the basic Kotlin types and the literals we use to create them:

- Numbers that are represented by types `Int`, `Long`, `Double`, `Float`, `Short`, and `Byte` are created with bare number values with possible suffixes for type customization. We can define negative numbers or decimal parts. We can also use underscores for nicer number formatting.
- Boolean values `true` and `false` are represented by the `Boolean` type.
- Characters, which are represented by the `Char` type. We define a character value using single quotation marks.
- Strings, which are used to represent text, are represented by the `String` type. Each string is just a

series of characters. We define strings inside double quotation marks.

So, we have the foundations for using Kotlin. Let's move on to more-complicated control structures that determine how our code behaves.

Conditional statements: if, when, try, and while

Most conditional statements, like the if-condition or the while-loop¹⁹, look the same in Kotlin, Java, C++, JavaScript, and most other modern languages. For instance, the if-statement is indistinguishable in all these languages:

```
if (predicate) {  
    // body  
}
```

However, the if-condition in Kotlin is more powerful and has capabilities that Kotlin's predecessors don't support. I assume that readers of this book have general experience in programming, therefore I won't explain how the if-condition or while-loop works. Instead, I will concentrate on the differences that Kotlin has introduced compared to other programming languages.

if-statement

Let's start with the aforementioned if-statement. It calls its body when its condition is satisfied (returns `true`). We can additionally add the `else` block, which is executed when the condition is not satisfied (returns `false`).

¹⁹As I described in the introduction, in this book, I decided to use a dash between "if", "when", "try", "while", "for", and the word describing it, like "condition", "loop", "statement" or "expression". I do this to improve readability. So, I will write "if-condition" instead of "if condition". "if" and "when" are conditions; "while" and "for" are loops. All of them can be used as statements, but only "if", "when" and "try" can be used as expressions. I decided not to use a dash after "if-else" or "if-else-if" and their descriptor.

```
fun main() {  
    val i = 1 // or 5  
    if (i < 3) { // i < 3 is used as a condition  
        // will be executed when condition returns true  
        println("Smaller")  
    } else {  
        // will be executed when condition returns false  
        println("Bigger")  
    }  
    // Prints Smaller if i == 1, or Bigger if i == 5  
}
```

One of Kotlin's superpowers is that an if-else statement can be used as an expression²⁰, therefore it produces a value.

```
val value = if (condition) {  
    // body 1  
} else {  
    // body 2  
}
```

What value is returned? For each body block, it is the result of the last statement (or `Unit` for an empty body or a statement that is not an expression²¹).

²⁰An expression in programming is a part of code that returns a value.

²¹`Unit` is an object without special meaning. It reminds me of Java's `Void`.

```
fun main() {
    var isOne = true
    val number1: Int = if (isOne) 1 else 0
    println(number1) // 1
    isOne = false
    val number2: Int = if (isOne) 1 else 0
    println(number2) // 0

    val superuser = true
    val hasAccess: Boolean = if (superuser) {
        println("Good morning, sir Admin")
        true
    } else {
        false
    }
    println(hasAccess) // true
}
```

When a body has only one statement, its result is the result of our if-else expression. In such a case, we don't need brackets.

```
val r: Int = if (one) 1 else 0
// a more readable alternative to
val r: Int = if (one) {
    1
} else {
    0
}
```

This way of using an if-statement is a Kotlin alternative to the Java or JavaScript ternary operator.

```
// Java
final String name = user == null ? "" : user.name
// JavaScript
const name = user === null ? "" : user.name
```

```
// Kotlin  
val name = if (user == null) "" else user.name
```

It should be said that if-else is longer than the ternary operator syntax. I believe this is the main reason why some developers want ternary operator syntax introduced in Kotlin. However, I am against this as if-else is a good replacement that is more readable and can be better formatted. Moreover, we have some additional Kotlin tools, which are also replacements for some ternary-operator use cases: the Elvis operator, extensions on nullable types (like `orEmpty`), or safe-calls. All these will be explained in detail in the chapter Nullability.

```
// Java  
String name = user == null ? "" : user.name  
  
// Kotlin  
val name = user?.name ?: ""  
// or  
val name = user?.name.orEmpty()
```

Notice that if you use the so-called if-else-if statement, it is just multiple connected if-else statements.

```
fun main() {  
    println("Is it going to rain?")  
    val probability = 70  
    if (probability < 40) {  
        println("Na-ha")  
    } else if (probability <= 80) {  
        println("Likely")  
    } else if (probability < 100) {  
        println("Yes")  
    } else {  
        println("Holly Crab")  
    }  
}
```

There is actually no such thing as an if-else-if expression: it is just one if-else expression inside another, as can be seen in strange cases where a method is executed on a whole if-else-if expression. Just take a look at the following puzzle and try to predict the result of this code.

```
// Function we can execute on any object, to print it
// 10.print() prints 10
// "ABC".print() prints ABC
fun Any?.print() {
    print(this)
}

fun printNumberSign(num: Int) {
    if (num < 0) {
        "negative"
    } else if (num > 0) {
        "positive"
    } else {
        "zero"
    }.print()
}

fun main(args: Array<String>) {
    printNumberSign(-2)
    print(",")
    printNumberSign(0)
    print(",")
    printNumberSign(2)
}
```

The answer is **not** “negative,zero,positive”, because there is no such thing as a single if-else-if expression (just two nested if-else expressions). So, the above `printNumberSign` implementation gives the same result as the following implementation.

```
fun printNumberSign(num: Int) {  
    if (num < 0) {  
        "negative"  
    } else {  
        if (num > 0) {  
            "positive"  
        } else {  
            "zero"  
        }.print()  
    }  
}
```

So, when we call `print` on the result, it is called on the result of the second if-else expression only (the one with “positive” and “zero”). This means that the code above will print “,zero,positive”. How can we fix this? We might use a bracket, but it is generally suggested that, instead of using if-else-if, we should use a when-statement when there is more than one condition. This can help avoid mistakes like the one in the puzzle above, and it makes code clearer and easier to read.

```
// Function we can execute on any object, to print it  
// 10.print() prints 10  
// "ABC".print() prints ABC  
fun Any?.print() {  
    print(this)  
}  
  
fun printNumberSign(num: Int) {  
    when {  
        num < 0 -> "negative"  
        num > 0 -> "positive"  
        else -> "zero"  
    }.print()  
}  
  
fun main(args: Array<String>) {  
    printNumberSign(-2) // negative  
    print(",") // ,
```

```
printNumberSign(0) // zero
print(",") // ,
printNumberSign(2) // positive
}
```

when-statement

The when-statement is an alternative to if-else-if. In every branch, we specify a predicate and the body that should be executed if this predicate returns `true` (and previous predicates did not). So, it works just like if-else-if but should be preferred because its syntax is better suited for multiple conditions.

```
fun main() {
    println("Is it going to rain?")
    val probability = 70
    when {
        probability < 40 -> {
            println("Na-ha")
        }
        probability <= 80 -> {
            println("Likely")
        }
        probability < 100 -> {
            println("Yes")
        }
        else -> {
            println("Holly Crab")
        }
    }
}
```

Like in an if-statement, braces are needed only for bodies with more than one statement.

```
fun main() {
    println("Is it going to rain?")
    val probability = 70
    when {
        probability < 40 -> println("Na-ha")
        probability <= 80 -> println("Likely")
        probability < 100 -> println("Yes")
        else -> println("Holly Crab")
    }
}
```

The when-statement can also be used as an expression because it can return a value. The result is the last expression of the chosen branch, therefore the following example will print "Likely".

```
fun main() {
    println("Is it going to rain?")
    val probability = 70
    val text = when {
        probability < 40 -> "Na-ha"
        probability <= 80 -> "Likely"
        probability < 100 -> "Yes"
        else -> "Holly Crab"
    }
    println(text)
}
```

The when-statement is often used as an expression body²²:

²²An expression body is a special syntax to implement function bodies with only one expression. This will be covered in the next chapter.

```
private fun getEmailErrorId(email: String) = when {
    email.isEmpty() -> R.string.error_field_required
    emailInvalid(email) -> R.string.error_invalid_email
    else -> null
}
```

when-statement with a value

There is also another form of the when-statement. If we add a value in brackets after the `when` keyword, then our when-statement becomes an alternative to the switch-case. However, it is a much more powerful alternative because it can not only compare values by equality, but it can also check if an object is of some type (using `is`), or if an object contains this value (using `in`). Each block can have multiple values we compare against, separated with a comma.

```
private val magicNumbers = listOf(7, 13)

fun describe(a: Any?) {
    when (a) {
        null -> println("Nothing")
        1, 2, 3 -> println("Small number")
        in magicNumbers -> println("Magic number")
        in 4..100 -> println("Big number")
        is String -> println("This is just $a")
        is Long, is Int -> println("This is Int or Long")
        else -> println("No idea, really")
    }
}

fun main() {
    describe(null) // Nothing
    describe(1) // Small number
    describe(3) // Small number
    describe(7) // Magic number
    describe(9) // Big number,
    // because 9 is in range from 4 to 100
}
```

```
    describe("AAA") // This is just AAA
    describe(1L) // This is Int or Long
    describe(-1) // This is Int or Long
    describe(1.0) // No idea, really,
    // because 1.0 is Double
}
```

The when-statement with a value can also be used as an expression because it can produce a value:

```
private val magicNumbers = listOf(7, 13)

fun describe(a: Any?): String = when (a) {
    null -> "Nothing"
    1, 2, 3 -> "Small number"
    in magicNumbers -> "Magic number"
    in 4..100 -> "Big number"
    is String -> "This is just $a"
    is Long, is Int -> "This is Int or Long"
    else -> "No idea, really"
}

fun main() {
    println(describe(null)) // Nothing
    println(describe(1)) // Small number
    println(describe(3)) // Small number
    println(describe(7)) // Magic number
    println(describe(9)) // Big number,
    // because 9 is in range from 4 to 100
    println(describe("AAA")) // This is just AAA
    println(describe(1L)) // This is Int or Long
    println(describe(-1)) // This is Int or Long
    println(describe(1.0)) // No idea, really,
    // because 1.0 is Double
}
```

Note that if we use a when-condition as an expression, its conditions must be exhaustive: it should cover all possible branch

conditions or provide an else branch, as in the example above. If not all conditions are covered, a compiler error is shown.

Kotlin does not support switch-case statements because we use the when-statement instead.

Inside the “when” parentheses, where we specify a value, we can also define a variable, and its value will be used in each condition.

```
fun showUsers() =  
    when (val response = requestUsers()) {  
        is Success -> showUsers(response.body)  
        is HttpError -> showException(response.exception)  
    }
```

is check

Since we have already mentioned the `is` operator, let's discuss it in a bit more depth. It checks if a value is of a certain type. We know already that `123` is of type `Int`, and `"ABC"` is of type `String`. Certainly, `123` is not of type `String`, and `"ABC"` is not of type `Int`. We can confirm this using the `is` check.

```
fun main() {  
    println(123 is Int) // true  
    println("ABC" is String) // true  
    println(123 is String) // false  
    println("ABC" is Int) // false  
}
```

Notice that `123` is an `Int`, but it is also a `Number`; the `is` check returns `true` for both these types.

```
fun main() {  
    println(123 is Int) // true  
    println(123 is Number) // true  
    println(3.14 is Double) // true  
    println(3.14 is Number) // true  
  
    println(123 is Double) // false  
    println(3.14 is Int) // false  
}
```

When we want to check if a value **is not** of a certain type, we can use **!is**, which is an equivalent to **is**-check, whose result value is negated.

```
fun main() {  
    println(123 !is Int) // false  
    println("ABC" !is String) // false  
    println(123 !is String) // true  
    println("ABC" !is Int) // true  
}
```

Explicit casting

You can always use a value whose type is `Int` as a `Number` because every `Int` is a `Number`. This process is known as *up-casting* because we change the value type from lower (more specific) to higher (less specific).

```
fun main() {  
    val i: Int = 123  
    val l: Long = 123L  
    val d: Double = 3.14  
  
    var number: Number = i // up-casting from Int to Number  
    number = l // up-casting from Long to Number  
    number = d // up-casting from Double to Number  
}
```

We can implicitly cast from a lower type to a higher one, but not the other way around. Every `Int` is a `Number`, but not every `Number` is an `Int` because there are more subtypes of `Number`, including `Double` or `Long`. This is why we cannot use `Number` where `Int` is expected. However, sometimes we have a situation where we are certain that a value is of a specified type, even though its supertype is used. Explicitly changing from a higher type to a lower type is called down-casting and requires the `as` operator in Kotlin.

```
var i: Number = 123

fun main() {
    val j = (i as Int) + 10
    println(j) // 133
}
```

In general, we avoid using `as` when not necessary because we consider it dangerous. Consider the above example. What if someone changes `123` to `3.14`? Both values are of type `Number`, so the code will compile without any problems or warnings. But `3.14` is `Double` not `Int`, and casting is not possible; therefore, the code above will break with an exception.

```
var i: Number = 3.14

fun main() {
    val j = (i as Int) + 10 // RUNTIME ERROR!
    println(j)
}
```

There are two ways to deal with this. The first is to use one of many Kotlin alternatives to cast our value safely. One example is using smart-casting, which will be described in the next section. Another example is a conversion function, like the `toInt` method, which transforms `Number` to `Int` (and possibly loses the decimal part).

```
var i: Number = 3.14

fun main() {
    val j = i.toInt() + 10
    println(j) // 13
}
```

The second option is the `as?` operator, which, instead of throwing an exception, returns `null` when casting is not possible. We will discuss handling nullable values later.

```
var n: Number = 123

fun main() {
    val i: Int? = n as? Int
    println(i) // 123
    val d: Double? = n as? Double
    println(d) // null
}
```

In Kotlin, we consider `as?` a safer option than `as`, but using both these operators too often is regarded as a code smell²³. Let's describe smart-casting, which is their popular alternative.

Smart-casting

Kotlin has a powerful feature called smart-casting, which allows automatic type casting when the compiler can be sure that a variable is of a certain type. Take a look at the following example:

²³I will use the term “code smell” to describe practices that are not explicitly wrong but should generally be avoided.

```
fun convertToInt(num: Number): Int =  
    if (num is Int) num // the type of num here is Int  
    else num.toInt()
```

The `convertToInt` function converts an argument of type `Number` to `Int` in the following way: if the argument is already of type `Int`, it is just returned; otherwise, it is converted using the `toInt` method. Notice that for this code to compile, the `num` inside the first body needs to be of type `Int`. In most languages, it needs to be cast, but this happens automatically in Kotlin. Take a look at another example:

```
fun lengthIfString(a: Any): Int {  
    if (a is String) {  
        return a.length // the type of a here is String  
    }  
    return 0  
}
```

Inside the `if`-condition predicate, we checked if `a` is of type `String`. The body of this statement will only be executed if the type check is successful. This is why `a` is of type `String` inside this body, which is why we can check its length. Such a conversion, from `Any` to `String`, is done implicitly by Kotlin. This can happen only when Kotlin is sure that no other thread can change our property, so when it is either a constant or a local variable. It will not work for non-local `var` properties because, in such cases, there is no guarantee that they have not been modified between check and usage (e.g., by another thread).

```
var obj: Any = "AAA"

fun main() {
    if (obj is String) {
        // println(obj.length) will not compile,
        // because `obj` can be modified by some
        // other thread, so Kotlin cannot be sure,
        // that at this point, is it still of type String
    }
}
```

Smart-casting is often used together with when-statements. When they are used together, they are sometimes referred to as “Kotlin type-safe pattern matching” because they can nicely cover the different possible types a value can be of. More examples will be presented when we discuss the sealed modifier.

```
fun handleResponse(response: Result<T>) {
    when (response) {
        is Success<*> -> showMessages(response.data)
        is Failure -> showError(response.throwable)
    }
}
```

While and do-while statements

The last important control structures we need to mention are while and do-while statements. Both look and work exactly the same as in Java, C++, and many other languages.

```
fun main() {
    var i = 1
    // while-statement
    while (i < 10) {
        print(i)
        i *= 2
    }
    // 1248

    var j = 1
    // do-while statement
    do {
        print(j)
        j *= 2
    } while (j < 10)
    // 1248
}
```

I hope they don't need any more explanation. When-statements and do-while-statements cannot be used as expressions. I will only add that both while and do-while statements are rarely used in Kotlin. Instead, we use collection or sequence processing functions, which will be covered in the Functional Kotlin book. For instance, the above code can be replaced with the following:

```
fun main() {
    generateSequence(1) { it * 2 }
        .takeWhile { it < 10 }
        .forEach(::print)
    // 1248
}
```

Summary

As you can see, Kotlin has introduced many powerful features to conditional statements. The if-condition and when-condition can be used as expressions. The when-statement is a more powerful alternative to if-else-if or switch-case. Type checks with smart-casting are supported. All these features make operating on nullable values both safe and pleasant, which makes the `null` value our friend, not an enemy. Now, let's see what Kotlin has changed in functions.

Functions

When Andrey Breslav, the initial Kotlin creator, was asked about his favorite feature during a discussion panel at Kotlin-Conf Amsterdam, he said it was functions²⁴. In the end, functions are our programs' most important building blocks. If you look at real-life applications, most of the code either defines or calls functions.

```
class AppsRepository(private val context: Context, private val prefs: AppPrefs) {

    private val excludeSystem get() = prefs.settings.excludeSystem
    private val excludeDisabled get() = prefs.settings.excludeDisabled
    private val excludeStore get() = prefs.settings.excludeStore
    private val ignoredApps get() = prefs.ignoredApps()

    private fun getPackageInfos(options: Int = 0): Sequence<PackageInfo> = runCatching {
        return context.packageManager.getInstalledPackages(options).asSequence()
            .filter { !excludeSystem || it.applicationInfo.flags and ApplicationInfo.FLAG_SYSTEM == 0 }
            .filter { !excludeDisabled || it.applicationInfo.flags and ApplicationInfo.FLAG_UPDATED_SYSTEM_APP == 0 }
            .filter { !excludeStore || it.applicationInfo.enabled }
            .filter { !excludeStore || !isAppStore(context.packageManager.getInstallerPackageName(it.packageName)) }
    }.getOrDefault {
        Log.e("AppsRepository", "getPackageInfos", it)
        return sequenceOf()
    }

    fun getPackageInfosFiltered(options: Int = 0) = getPackageInfos(options).filter { !ignoredApps.contains(it.packageName) }

    fun getApps(options: Int = 0) = getPackageInfos(options).mapIndexed { i, app ->
        AppInstalled(
            i,
            app.name(context),
            app.packageName,
            app.versionName ?: "",
            app.versionCode,
            iconUri(app.packageName, app.applicationInfo.icon),
            ignoredApps.contains(app.packageName)
        )
    }.sortedBy { it.name }.sortedBy { it.ignored }.toList()

    fun getAppsFiltered(apps: Sequence<PackageInfo>) = apps.mapIndexed { i, app ->
        AppInstalled(
            i,
            app.name(context),
            app.packageName,
            app.versionName ?: "",
            app.versionCode,
            iconUri(app.packageName, app.applicationInfo.icon),
            ignoredApps.contains(app.packageName)
        )
    }.sortedBy { it.name }.sortedBy { it.ignored }

    // Checks if Play Store or Amazon Store
    private fun isAppStore(name: String?) = name?.contains("com.android.vending")?.orFalse() || name?.contains("com.amazon")?.orFalse()
}
```

As an example, I used a random class from the APKUpdater open-source project. Notice that nearly every line either defines or calls a function.

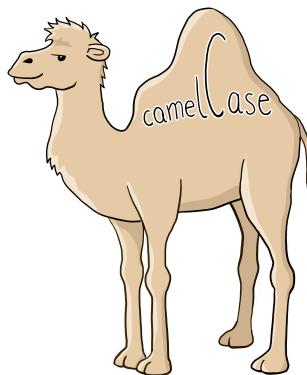
In Kotlin, we define functions using the `fun` keyword. This is why we have so much `fun` in Kotlin. With a bit of creativity, a function can consist only of `fun`:

²⁴Source: <https://youtu.be/heqjfks4z2I?t=660>

```
fun <Fun> `fun`(`fun`: Fun): Fun = `fun`
```

This is the so-called *identity function*, a function that returns its argument without any modifications. It has a generic type parameter `Fun`, but this will be explained in the chapter *Generics*.

By convention, we name functions using lower camelCase syntax²⁵. Formally, we can use characters, underscore `_`, and numbers (but not at the first position), but in general just characters should be used.



In Kotlin, we name functions with `lowerCamelCase`.

This is what a typical function looks like:

²⁵This rule has some exceptions. For example, on Android, Jetpack Compose functions should be named using Upper-CamelCase by convention. Also, unit tests are often named with full sentences inside braces.

```
fun square(x: Double): Double {
    return x * x
}

fun main() {
    println(square(10.0)) // 100.0
}
```

Notice that the parameter type is specified after the variable name and a colon, and the result type is specified after the parameter brackets and a colon. Such notation is typical of languages with powerful support for type inference because it is easier to add or remove explicit type definitions.

```
val a: Int = 123
// easy to transform from or to
val a = 123

fun add(a: Int, b: Int): Int = a + b

// easy to transform from or to
fun add(a: Int, b: Int) = a + b
```

To use a reserved keyword as a function name (like `fun` or `when`), use backticks, as in the example above. When a function has an illegal name, both its definition and calls require backticks.

Another use case for backticks is naming unit-test functions so that they can be described in plain English, as in the example below. This is not standard practice, but it is still quite a popular practice that many teams choose to adopt.

```
class CartViewModelTests {
    @Test
    fun `should show error dialog when no items loaded`() {
        ...
    }
}
```

Single-expression functions

Many functions in real-life projects just have a single expression²⁶, so they start and immediately use the `return` keyword. The `square` function defined above is a great example. For such functions, instead of defining the body with braces, we can use the equality sign (=) and just specify the expression that calculates the result without specifying `return`. This is **single-expression syntax**, and functions that use it are called **single-expression functions**.

```
fun square(x: Double): Double = x * x

fun main() {
    println(square(10.0)) // 100.0
}
```

An expression can be more complicated and take multiple lines. This is fine as long as its body is a single statement.

```
fun findUsers(userFilters: UserFilters): List<User> =
    userRepository
        .getUsers()
        .map { it.toDomain() }
        .filter { userFilters.accepts(it) }
```

When we use single-expression function syntax, we can infer the result type. We don't need to, as explicit result type might still be useful for safety and readability²⁷, but we can.

²⁶As a reminder, an expression is a part of our code that returns a value.

²⁷See Effective Kotlin Item 4: Do not expose inferred types

```
fun square(x: Double) = x * x

fun main() {
    println(square(10.0)) // 100.0
}
```

Functions on all levels

Kotlin allows us to define functions on many levels, but this isn't very obvious as Java only allows functions inside classes. In Kotlin, we can define:

- functions in files outside any classes, called **top-level functions**,
- functions inside classes or objects, called **member functions** (they are also **methods**),
- functions inside functions, called **local functions** or **nested functions**.

```
// Top-level function
fun double(i: Int) = i * 2

class A {
    // Member function (method)
    private fun triple(i: Int) = i * 3

    // Member function (method)
    fun twelveTimes(i: Int): Int {
        // Local function
        fun fourTimes() = double(double(i))
        return triple(fourTimes())
    }
}

// Top-level function
fun main(args: Array<String>) {
    double(1) // 2
```

```
A().twelveTimes(2) // 24  
}
```

Top-level functions (defined outside classes) are often used to define utils - small but useful functions that help us with development. Top-level functions can be moved and split across files. In many cases, top-level functions in Kotlin are better than static functions in Java. Using them seems intuitive and convenient for developers.

However, it's a different story with local functions (defined inside functions). I often see that developers lack the imagination to use them (due to lack of exposure to them). Local functions are popular in JavaScript and Python, but there's nothing like this in Java. The power of local functions is that they can directly access or modify local variables. They are used to extract repetitive code inside a function that operates on local variables. Longer functions should tell a "story", and local subroutines can wrap a block expression in a descriptive name.

Take a look at the below example, which presents a function that validates a form. It checks conditions for the form fields. If a condition is not matched, we should show an error and change the local variable `isValid` to `false`, in which case we should not return from the function because we want to check all the fields (we should not stop at the first one that fails). This is an example of where a local function can help us extract repetitive behavior.

```
fun validateForm() {  
    var isValid = true  
    val errors = mutableListOf<String>()  
    fun addError(view: FormView, error: String) {  
        view.error = error  
        errors += error  
        isValid = false  
    }  
  
    val email = emailView.text
```

```
if (email.isBlank()) {
    addError(emailView, "Email cannot be empty or blank")
}

val pass = passView.text.trim()
if (pass.length < 3) {
    addError(passView, "Password too short")
}

if (isValid) {
    tryLogin(email, pass)
} else {
    showErrors(errors)
}
}
```

Parameters and arguments

A variable defined as a part of a function definition is called a **parameter**. The value that is passed when we call a function is called an **argument**.

```
fun square(x: Double) = x * x // x is a parameter

fun main() {
    println(square(10.0)) // 10.0 is an argument
    println(square(0.0)) // 0.0 is an argument
}
```

In Kotlin, parameters are read-only, so we cannot reassign their value.

```
fun a(i: Int) {
    i = i + 10 // ERROR
    // ...
}
```

If you need to modify a parameter variable, the only way is to shadow it with a local variable that is mutable.

```
fun a(i: Int) {  
    var i = i + 10  
    // ...  
}
```

This is possible but discouraged. A parameter holds a value that was used as an argument, and this value should not change. A local variable that can be read-write represents a different concept and should therefore have a different name.

Unit return type

In Kotlin, all functions have a result type, so every function call is an expression. When a type is not specified, the default result type is `Unit`, and the default result value is the `Unit` object.

```
fun someFunction() {}  
  
fun main() {  
    val res: Unit = someFunction()  
    println(res) // kotlin.Unit  
}
```

`Unit` is just a very simple object that is used as a placeholder when nothing else is returned. When you specify a function without an explicit result type, its result type will implicitly be `Unit`. When you define a function without `return` in the last line, it is the same as using `return` with no value. Using `return` with no value is the same as returning `Unit`.

```
fun a() {}

// the same as
fun a(): Unit {}

// the same as
fun a(): Unit {
    return
}

// the same as
fun a(): Unit {
    return Unit
}
```

Vararg parameters

Each parameter expects one argument, except for parameters marked with the `vararg` modifier. Such parameters accept any number of arguments.

```
fun a(vararg params: Int) {}

fun main() {
    a()
    a(1)
    a(1, 2)
    a(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
}
```

A good example of such a function is `listof`, which produces a list from values used as arguments.

```
fun main() {  
    println(listOf(1, 3, 5, 6)) // [1, 3, 5, 6]  
    println(listOf("A", "B", "C")) // [A, B, C]  
}
```

This means a vararg parameter holds a collection of values, therefore it cannot have the type of a single object. So, the vararg parameter represents an array of the declared type, and we can iterate over arrays using a for loop (which will be explained in more depth in the next chapter).

```
fun concatenate(vararg strings: String): String {  
    // The type of `strings` is Array<String>  
    var accumulator = ""  
    for (s in strings) accumulator += s  
    return accumulator  
}  
  
fun sum(vararg ints: Int): Int {  
    // The type of `ints` is IntArray  
    var accumulator = 0  
    for (i in ints) accumulator += i  
    return accumulator  
}  
  
fun main() {  
    println(concatenate()) //  
    println(concatenate("A", "B")) // AB  
    println(sum()) // 0  
    println(sum(1, 2, 3)) // 6  
}
```

We will get back to vararg parameters in the chapter *Collections*, in the section dedicated to arrays.

Named parameter syntax and default arguments

When we declare functions, we often specify optional parameters. A good example is `joinToString`, which transforms an iterable into a `String`. It can be used without any arguments, or we might change its behavior with concrete arguments.

```
fun main() {  
    val list = listOf(1, 2, 3, 4)  
    println(list.joinToString()) // 1, 2, 3, 4  
    println(list.joinToString(separator = "-")) // 1-2-3-4  
    println(list.joinToString(limit = 2)) // 1, 2, ...  
}
```

Many more functions in Kotlin use optional parametrization, but how is this done? It is enough to place an equality sign after a parameter and then specify the default value.

```
fun cheer(how: String = "Hello,", who: String = "World") {  
    println("$how $who")  
}  
  
fun main() {  
    cheer() // Hello, World  
    cheer("Hi") // Hi World  
}
```

Values specified this way are created on-demand when there is no parameter for their position. This is not Python, therefore they are not stored statically, which is why it's safe to use mutable values as default arguments.

```
fun addOneAndPrint(list: MutableList<Int> = mutableListOf()) {
    list.add(1)
    println(list)
}

fun main() {
    addOneAndPrint() // [1]
    addOneAndPrint() // [1]
    addOneAndPrint() // [1]
}
```

In Python, the analogous code would produce [1], [1, 1], and [1, 1, 1].

When we call a function, we can specify an argument's position by its parameter name, like in the example below. This way, we can specify later optional positions without specifying previous ones. This is called *named parameter syntax*.

```
fun cheer(how: String = "Hello,", who: String = "World") {
    print("$how $who")
}

fun main() {
    cheer(who = "Group") // Hello, Group
}
```

Named parameter syntax is very useful for improving our code's readability. When an argument's meaning is not clear, it is better to specify a parameter name for it.

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    println(list.joinToString("-")) // 1-2-3-4
    // better
    println(list.joinToString(separator = "-")) // 1-2-3-4
}
```

Naming arguments also prevents mistakes that are a result of changing parameter positions.

```
class User(  
    val name: String,  
    val surname: String,  
)  
  
val user = User(  
    name = "Norbert",  
    surname = "Moskała",  
)
```

In the above example, without named arguments a developer might flip the `name` and `surname` positions; if named arguments were not used here, this would lead to an incorrect name and surname in the object. Named arguments protect us from such situations.

It is considered a good practice to use the named arguments convention when we call functions with many arguments, some of whose meanings might not be obvious to developers reading our code in the future.

Function overloading

In Kotlin, we can define functions with the same name in the same scope (file or class) as long as they have different parameter types or a different number of parameters. This is known as function **overloading**. Kotlin decides which function to execute based on the types of the specified arguments.

```
fun a(a: Any) = "Any"  
fun a(i: Int) = "Int"  
fun a(l: Long) = "Long"  
  
fun main() {  
    println(a(1)) // Int  
    println(a(18L)) // Long  
    println(a("ABC")) // Any  
}
```

A practical example of function overloading is providing multiple function variants for user convenience.

```
import java.math.BigDecimal

class Money(val amount: BigDecimal, val currency: String)

fun pln(amount: BigDecimal) = Money(amount, "PLN")
fun pln(amount: Int) = pln(amount.toBigDecimal())
fun pln(amount: Double) = pln(amount.toBigDecimal())
```

Infix syntax

Methods with a single parameter can use the `infix` modifier, which allows a special kind of function call: without the dot and the argument parentheses.

```
class View
class ViewInteractor {
    infix fun clicks(view: View) {
        // ...
    }
}

fun main() {
    val aView = View()
    val interactor = ViewInteractor()

    // regular notation
    interactor.clicks(aView)
    // infix notation
    interactor clicks aView
}
```

This notation is used by some functions from Kotlin stdlib (Standard Library), like the `and`, `or` and `xor` bitwise operations on numbers (presented in the chapter *Basic types, their literals and operations*).

```
fun main() {  
    // infix notation  
    println(0b011 and 0b001) // 1, that is 0b001  
    println(0b011 or 0b001) // 3, that is 0b011  
    println(0b011 xor 0b001) // 2, that is 0b010  
  
    // regular notation  
    println(0b011.and(0b001)) // 1, that is 0b001  
    println(0b011.or(0b001)) // 3, that is 0b011  
    println(0b011.xor(0b001)) // 2, that is 0b010  
}
```

Infix notation is only for our convenience. It is an example of Kotlin syntactic sugar - syntax that is designed only to make things easier to read or express.

Regarding the position of operators or functions in relation to their operands or arguments, we use three kinds of position types: prefix, infix, and postfix. Prefix notation is when we place the operator or function **before** the operands or arguments²⁸. A good example is a plus or minus placed before a single number (like `+12` or `-3.14`). One might argue that a top-level function call also uses prefix notation because the function name comes before the arguments (like `maxOf(10, 20)`). Infix notation is when we place the operator or function **between** the operands or arguments²⁹. A good example is a plus or minus between two numbers (like `1 + 2` or `10 - 7`). One might argue that a method call with arguments also uses infix notation because the function name comes between the receiver (the object we call this method

²⁸From the Latin word *praefixus*, which means “fixed in front”.

²⁹From the Latin word *infixus*, the past participle of *infigere*, which we might translate as “fixed in between”.

on) and arguments (like `account.add(money)`). In Kotlin, we use the term “infix notation” more restrictively to reference the special notation we use for methods with the `infix` modifier. Postfix notation is when we place the operator or function **after** the operands or arguments³⁰. In modern programming, postfix notation is practically not used anymore. One might argue that calling a method with no arguments is postfix notation, as in `str.uppercase()`.

Function formatting

When a function declaration (name, parameters, and result type) is too long to fit in a single line, we split it such that every parameter definition is on a different line, and the beginning and end of the function declaration are also on separate lines.

```
fun veryLongFunction(  
    param1: Param1Type,  
    param2: Param2Type,  
    param3: Param3Type,  
): ResultType {  
    // body  
}
```

Classes are formatted in the same way³¹:

³⁰Made from the prefix “post-“, which means “after, behind”, and the word “fix”, meaning “fixed in place”.

³¹We will discuss classes later in this book, in the chapter *Classes and interfaces*.

```
class VeryLongClass(  
    val property1: Type1,  
    val property2: Type2,  
    val property3: Type3,  
) : ParentClass(), Interface1, Interface2 {  
    // body  
}
```

When a function call³² is too long, we format it similarly: each argument is on a different line. However, there are exceptions to this rule, such as keeping multiple vararg parameters on the same line.

```
fun makeUser(  
    name: String,  
    surname: String,  
) : User = User(  
    name = name,  
    surname = surname,  
)  
  
class User(  
    val name: String,  
    val surname: String,  
)  
  
fun main() {  
    val user = makeUser(  
        name = "Norbert",  
        surname = "Moskała",  
    )  
  
    val characters = listOf(  
        "A", "B", "C", "D", "E", "F", "G", "H", "I", "J",  
        "K", "L", "M", "N", "O", "P", "R", "S", "T", "U",  
        "W", "X", "Y", "Z",  
    )
```

³²A constructor call is also considered a function call in Kotlin.

```
)  
}
```

In this book, the width of my lines is much smaller than in regular projects, so I am forced to break lines much more often than I would like to.

Notice that when I specify arguments or parameters, I sometimes add a comma at the end. This is a so-called **trailing comma**. Such notation is optional.

```
fun printName(  
    name: String,  
    surname: String, // <- trailing comma  
) {  
    println("$name $surname")  
}  
  
fun main() {  
    printName(  
        name = "Norbert",  
        surname = "Moskała", // <- trailing comma  
    )  
}
```

I like using trailing comma notation because it makes it easier to add another element in the future. Without it, adding or removing an element requires not only a new line but also an additional comma after the last element. This leads to meaningless line modifications on Git, which makes it harder to read what has actually changed in our project. Some developers don't like trailing comma notation, which can sometimes lead to a holy war. Decide in your team if you like it or not, and be consistent in your projects.

```

3   fun printName(
4     name: String,
5     surname: String,
6   ) {
7   -   println("$name $surname")
8 }
9
10 fun main() {
11   printName(
12     name = "Norbert",
13     surname = "Moskała",
14   )
15 } ⊖

```

```

3   fun printName(
4     name: String,
5     surname: String,
6   +   middleName: String? = null,
7   ) {
8   +   if (middleName != null) {
9   +     println("$name $middleName $surname")
10 +   } else {
11   +     println("$name $surname")
12   +   }
13 }
14
15 fun main() {
16   printName(
17     name = "Norbert",
18     surname = "Moskała",
19   +   middleName = "Jan",
20   )
21 } ⊖

```

Adding a parameter and an argument on git when a trailing comma is used.

```

2
3   fun printName(
4     name: String,
5   -   surname: String
6   ) {
7   -   println("$name $surname")
8 }
9
10 fun main() {
11   printName(
12     name = "Norbert",
13   -   surname = "Moskała"
14   )
15 } ⊖

```

```

2
3   fun printName(
4     name: String,
5   +   surname: String,
6   +   middleName: String? = null
7   ) {
8   +   if (middleName != null) {
9   +     println("$name $middleName $surname")
10 +   } else {
11   +     println("$name $surname")
12   +   }
13 }
14
15 fun main() {
16   printName(
17     name = "Norbert",
18   +   surname = "Moskała",
19   +   middleName = "Jan"
20   )
21 } ⊖

```

Adding a parameter and an argument on git when a trailing comma is not used.

Summary

As you can see, functions in Kotlin have a lot of powerful features. Single-expression syntax makes simple functions shorter. Named and default arguments help us improve safety and readability. The `Unit` result type makes every function call an expression. Vararg parameters allow any number of arguments to be used for one parameter position. Infix notation introduces a more convenient way to call certain kinds of functions. Trailing commas minimize the number of changes on git. All this is for our convenience. For now though, let's move on to another topic: using a for-loop.

The power of the for-loop

In Java and other older languages, a for-loop typically has three parts: the first initializes the variable before the loop starts; the second contains the condition for the execution of the code block; the third is executed after the code block.

```
// Java
for(int i=0; i < 5; i++){
    System.out.println(i);
}
```

However, this is considered complicated and error-prone. Just consider a situation in which someone uses `>` or `<=` instead of `<`. Such a small difference is not easy to notice, but it essentially influences the behavior of this for-loop.

As an alternative to this classic for-loop, many languages have introduced a modern alternative for iterating over collections. This is why, in languages like Java or JavaScript, there are two completely different kinds of for-loops, both of which are defined with the same `for` keyword. Kotlin has simplified this. In Kotlin, we have one universal for-loop that can be expressively used to iterate over a collection, a map, a range of numbers, and much more.

In general, a for-loop is used in Kotlin to iterate over something that is iterable³³.

³³Has the `iterator` operator method.

A hand-drawn diagram illustrating the structure of a for-loop. The code is:

```
for (value in iterable) {  
    println(value)  
}
```

Annotations above the code explain its components:

- The word "variable name" has a curly arrow pointing to the identifier "value".
- The word "object we iterate over" has a curly arrow pointing to the identifier "iterable".
- The brace closing the loop has a handwritten label "body" written vertically next to it.

We can iterate over lists or sets.

```
fun main() {  
    val list = listOf("A", "B", "C")  
    for (letter in list) {  
        print(letter)  
    }  
  
    // Variable type can be explicit  
    for (str: String in setOf("D", "E", "F")) {  
        print(str)  
    }  
}  
// ABCDEF
```

We can also iterate over any other object as long as it contains the `iterator` method with no parameters, plus the `Iterator` result type and the `operator` modifier. The easiest way to define this method is to make your class implement the `Iterable` interface.

```
fun main() {
    val tree = Tree(
        value = "B",
        left = Tree("A"),
        right = Tree("D", left = Tree("C"))
    )

    for (value in tree) {
        print(value) // ABCD
    }
}

class Tree(
    val value: String,
    val left: Tree? = null,
    val right: Tree? = null,
) : Iterable<String> {

    override fun iterator(): Iterator<String> = iterator {
        if (left != null) yieldAll(left)
        yield(value)
        if (right != null) yieldAll(right)
    }
}
```

The inferred variable type of the variable defined inside the for-loop comes from the `Iterable` type argument. When we iterate over `Iterable<User>`, the inferred element type will be `User`. When we iterate over `Iterable<Long?>`, the inferred element type will be `Long?`. The same applies to all other types.

This mechanism, which relies on `Iterable`, is really powerful and allows us to cover numerous use cases, one of the most notable of which is the use of `ranges` to express progressions.

Ranges

In Kotlin, if you place two dots between two numbers, like `1..5`, you create an `IntRange`. This class implements

Iterable<Int>, so we can use it in a for-loop:

```
fun main() {  
    for (i in 1..5) {  
        print(i)  
    }  
}  
// 12345
```

This solution is efficient as well as convenient because the Kotlin compiler optimizes its performance under the hood.

Ranges created with .. include the last value (which means they are **closed ranges**). If you want a range that stops before the last value, use the until infix function instead.

```
fun main() {  
    for (i in 1 until 5) {  
        print(i)  
    }  
}  
// 1234
```

Both .. and until start with the value on their left and progress toward the right number in increments of one. If you use a bigger number on the left, the result is an empty range.

```
fun main() {  
    for (i in 5..1) {  
        print(i)  
    }  
    for (i in 5 until 1) {  
        print(i)  
    }  
}  
// (nothing is printed)
```

If you want to iterate in the other direction, from larger to smaller numbers, use the downTo function.

```
fun main() {
    for (i in 5 downTo 1) {
        print(i)
    }
}
// 54321
```

The default step in all those cases is 1. If you want to use a different step, you should use the `step` infix function.

```
fun main() {
    for (i in 1..10 step 3) {
        print("$i ")
    }
    println()
    for (i in 1 until 10 step 3) {
        print("$i ")
    }
    println()
    for (i in 10 downTo 1 step 3) {
        print("$i ")
    }
}
// 1 4 7 10
// 1 4 7
// 10 7 4 1
```

Break and continue

Inside loops, we can use the `break` and `continue` keywords:

- `break` - terminates the nearest enclosing loop.
- `continue` - proceeds to the next step of the nearest enclosing loop.

```
fun main() {
    for (i in 1..5) {
        if (i == 3) break
        print(i)
    }
    // 12

    println()

    for (i in 1..5) {
        if (i == 3) continue
        print(i)
    }
    // 1245
}
```

Both are used rather rarely, and I had trouble finding even one real-life example in the commercial projects I have co-created. I also assume that they are well-known to developers who've come to Kotlin from older languages. This is why I present these keywords so briefly.

Use cases

Developers with experience in older languages often use a for-loop where slightly more-modern alternatives should be used instead. For instance, in some projects I can find a for-loop that is used to iterate over elements with indices.

```
fun main() {
    val names = listOf("Alex", "Bob", "Celina")

    for (i in 0 until names.size) {
        val name = names[i]
        println("[\$i] \$name")
    }
}
// [0] Alex
```

```
// [1] Bob  
// [2] Celina
```

This is not a good solution. There are multiple ways to do this better in Kotlin.

First, instead of explicitly iterating over a range `0 until names.size`, we could use the `indices` property, which returns a range of available indices.

```
fun main() {  
    val names = listOf("Alex", "Bob", "Celina")  
  
    for (i in names.indices) {  
        val name = names[i]  
        println("[\$i] \$name")  
    }  
}  
// [0] Alex  
// [1] Bob  
// [2] Celina
```

Second, instead of iterating over indices and finding an element for each of them, we could instead iterate over indexed values. We can create indexed values using `withIndex` on iterable. Each indexed value includes both an index and a value. Such objects can be destructured in a for-loop³⁴.

³⁴Destructuring will be explained in more depth in the Data classes chapter.

```
fun main() {
    val names = listOf("Alex", "Bob", "Celina")

    for ((i, name) in names.withIndex()) {
        println("[\$i] \$name")
    }
}

// [0] Alex
// [1] Bob
// [2] Celina
```

Third, an even better solution is to use `forEachIndexed`, which is explained in the next book: *Functional Kotlin*.

```
fun main() {
    val names = listOf("Alex", "Bob", "Celina")

    names.forEachIndexed { i, name ->
        println("[\$i] \$name")
    }
}

// [0] Alex
// [1] Bob
// [2] Celina
```

Another popular use case is iterating over a map. Developers with a Java background often do it this way:

```
fun main() {
    val capitals = mapOf(
        "USA" to "Washington DC",
        "Poland" to "Warsaw",
        "Ukraine" to "Kiev"
    )

    for (entry in capitals.entries) {
        println(
            "The capital of \${entry.key} is \${entry.value}")
    }
}
```

```
        }
    }
// The capital of USA is Washington DC
// The capital of Poland is Warsaw
// The capital of Ukraine is Kiev
```

This can be improved by directly iterating over a map, so calling `entries` is unnecessary. Also, we can destructure entries to better name the values.

```
fun main() {
    val capitals = mapOf(
        "USA" to "Washington DC",
        "Poland" to "Warsaw",
        "Ukraine" to "Kiev"
    )

    for ((country, capital) in capitals) {
        println("The capital of $country is $capital")
    }
}
// The capital of USA is Washington DC
// The capital of Poland is Warsaw
// The capital of Ukraine is Kiev
```

We can use `forEach` for a map.

```
fun main() {
    val capitals = mapOf(
        "USA" to "Washington DC",
        "Poland" to "Warsaw",
        "Ukraine" to "Kiev"
    )

    capitals.forEach { (country, capital) ->
        println("The capital of $country is $capital")
    }
}
```

```
// The capital of USA is Washington DC  
// The capital of Poland is Warsaw  
// The capital of Ukraine is Kiev
```

Summary

In this chapter, we've learned about using the for-loop. It is really simple and powerful in Kotlin, so it's worth knowing how it works, even though it's not used very often (due to Kotlin's amazing functional features, which are often used instead).

Now, let's talk about one of the most important Kotlin improvements over Java: good support for handling nullability.

Nullability

Kotlin started as a remedy to Java problems, and the biggest problem in Java is nullability. In Java, like in many other languages, every variable is nullable, so it might have the `null` value. Every call on a `null` value leads to the famous `NullPointerException` (NPE). This is the #1 exception in most Java projects³⁵. It is so common that it is often referred to as “the billion dollar mistake” after the famous speech by Sir Charles Antony Richard Hoare, where he said: “I call it my billion-dollar mistake. It was the invention of the null reference in 1965... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years”.

One of Kotlin’s priorities was to solve this problem finally, and it achieved this perfectly. The mechanisms introduced in Kotlin are so effective that seeing `NullPointerException` thrown from Kotlin code is extremely rare. The `null` value stopped being a problem, and Kotlin developers are no longer scared of it. It has become our friend.

So, how does nullability work in Kotlin? Everything is based on a few rules:

1. Every property needs to have an explicit value. There is no such thing as an implicit `null` value.

```
var person: Person // COMPILATION ERROR,  
// the property needs to be initialized
```

2. A regular type does not accept the `null` value.

```
var person: Person = null // COMPILATION ERROR,  
// Person is not a nullable type, and cannot be `null`
```

³⁵Some research confirms this: for example, data collected by OverOps confirms that `NullPointerException` is the most common exception in 70% of production environments.

3. To specify a nullable type, you need to end a regular type with a question mark (?).

```
var person: Person? = null // OK
```

4. Nullable values cannot be used directly. They must be used safely or cast first (using one of the tools presented later in this chapter).

```
person.name // COMPILE ERROR,  
// person type is nullable, so we cannot use it directly
```

Thanks to all these mechanisms, we always know what can be `null` and what cannot. We use nullability only when we need it - when there is a reason to. In such cases, users are forced to explicitly handle this nullability. In all other cases, there is no need to do this. This is a perfect solution, but good tools are needed to deal with nullability in a way that's convenient for developers.

Kotlin supports a variety of ways of using a nullable value, including safe calls, not-null assertions, smart-casting, or the Elvis operator. Let's discuss these one by one.

Safe calls

The simplest way to call a method or a property on a nullable value is with a safe call, which is a question mark and a dot (`?.`) instead of just a regular dot (`.`). Safe calls work as follows:

- if a value is `null`, it does nothing and returns `null`,
- if a value is not `null`, it works like a regular call.

```
class User(val name: String) {
    fun cheer() {
        println("Hello, my name is $name")
    }
}

var user: User? = null

fun main() {
    user?.cheer() // (does nothing)
    println(user?.name) // null
    user = User("Cookie")
    user?.cheer() // Hello, my name is Cookie
    println(user?.name) // Cookie
}
```

Notice that the result of a safe call is always a nullable type because a safe call returns `null` when it is called on a `null`. This means that nullability propagates. If you need to find out the length of a user's name, calling `user?.name.length` will not compile. Even though `name` is not nullable, the result of `user?.name` is `String?`, so we need to use a safe call again: `user?.name?.length`.

```
class User(val name: String) {
    fun cheer() {
        println("Hello, my name is $name")
    }
}

var user: User? = null

fun main() {
    // println(user?.name.length) // ILLEGAL
    println(user?.name?.length) // null
    user = User("Cookie")
    // println(user?.name.length) // ILLEGAL
    println(user?.name?.length) // 6
}
```

Not-null assertion

When we don't expect a `null` value, and we want to throw an exception if one occurs, we can use the not-null assertion `!!`.

```
class User(val name: String) {
    fun cheer() {
        println("Hello, my name is $name")
    }
}

var user: User? = User("Cookie")

fun main() {
    println(user!!.name.length) // 6
    user = null
    println(user!!.name.length) // throws NullPointerException
}
```

This is not a very safe option because if we are wrong and a `null` value is where we don't expect it, this leads to a `NullPointerException`. Sometimes we want to throw an exception to ensure that there are no situations where `null` is used, but we generally prefer to throw a more meaningful exception³⁶. For this, the most popular options are:

- `requireNotNull`, which accepts a nullable value as an argument and throws `IllegalArgumentException` if this value is null. Otherwise, it returns this value as non-nullable.
- `checkNotNull`, which accepts a nullable value as an argument and throws `IllegalStateException` if this value is null. Otherwise, it returns this value as non-nullable.

³⁶There is more about exceptions, `IllegalArgumentException` and `IllegalStateException` in the chapter Exceptions.

```
fun sendData(dataWrapped: Wrapper<Data>) {
    val data = requireNotNull(dataWrapped.data)
    val connection = checkNotNull(connections["db"])
    connection.send(data)
}
```

Smart-casting

Smart-casting also works for nullability. Therefore, in the scope of a non-nullability check, a nullable type is cast to a non-nullable type.

```
fun printLengthIfNotNull(str: String?) {
    if (str != null) {
        println(str.length) // str smart-casted to String
    }
}
```

Smart-casting also works when we use `return` or `throw` if a value is not `null`.

```
fun printLengthIfNotNull(str: String?) {
    if (str == null) return
    println(str.length) // str smart-casted to String
}
```

```
fun printLengthIfNotNullOrThrow(str: String?) {
    if (str == null) throw Error()
    println(str.length) // str smart-casted to String
}
```

Smart-casting is quite smart and works in different cases, such as after `&&` and `||` in logical expressions.

```
fun printLengthIfNotNull(str: String?) {
    if (str != null && str.length > 0) {
        // str in expression above smart-casted to String
        // ...
    }
}

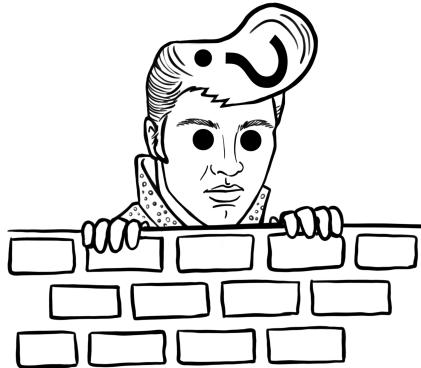
fun printLengthIfNotNull(str: String?) {
    if (str == null || str.length == 0) {
        // str in expression above smart-casted to String
        // ...
    }
}

fun printLengthIfNotNullOrThrow(str: String?) {
    requireNotNull(str) // str smart-casted to String
    println(str.length)
}
```

Smart-casting works in the code above thanks to a feature called **contracts**, which is explained in the book *Advanced Kotlin*.

The Elvis operator

The last special Kotlin feature that is used to support nullability is the Elvis operator `?:`. Yes, it is a question mark and a colon. It is called the Elvis operator because it looks like Elvis Presley (with his characteristic hair), looking at us from behind a wall, so we can only see his hair and eyes.



It is placed between two values. If the value on the left side of the Elvis operator is not `null`, we use the nullable value that results from the Elvis operator. If the left side is `null`, then the right side is returned.

```
fun main() {  
    println("A" ?: "B") // A  
    println(null ?: "B") // B  
    println("A" ?: null) // A  
    println(null ?: null) // null  
}
```

We can use the Elvis operator to provide a default value for nullable values.

```
class User(val name: String)

fun printName(user: User?) {
    val name: String = user?.name ?: "default"
    println(name)
}

fun main() {
    printName(User("Cookie")) // Cookie
    printName(null) // default
}
```

Extensions on nullable types

Regular functions cannot be called on nullable variables. However, there is a special kind of function that can be defined such that it can be called on nullable variables³⁷. Thanks to this, Kotlin stdlib defines the following functions that can be called on `String? :`

- `orEmpty` returns the value if it is not `null`. Otherwise it returns an empty string.
- `isNullOrEmpty` returns `true` if the value is `null` or empty. Otherwise, it returns `false`.
- `isNullOrBlank` returns `true` if the value is `null` or blank. Otherwise, it returns `false`.

³⁷These are extension functions, which we will discuss in the chapter Extensions.

```
fun check(str: String?) {
    println("The value: \$str")
    println("The value or empty: \$str.orEmpty()")
    println("Is null or empty? " + str.isNullOrEmpty())
    println("Is null or blank? " + str.isNullOrBlank())
}

fun main() {
    check("ABC")
    // The value: "ABC"
    // The value or empty: "ABC"
    // Is null or empty? false
    // Is null or blank? false
    check(null)
    // The value: "null"
    // The value or empty: ""
    // Is null or empty? true
    // Is null or blank? true
    check("")
    // The value: ""
    // The value or empty: ""
    // Is null or empty? true
    // Is null or blank? true
    check("      ")
    // The value: "      "
    // The value or empty: "      "
    // Is null or empty? false
    // Is null or blank? true
}
```

Kotlin stdlib also includes similar functions for nullable lists:

- `orEmpty` returns the value if it is not `null`. Otherwise, it returns an empty list .
- `isNullOrEmpty` returns `true`, returns `true` if the value is `null` or empty. Otherwise, it returns `false`.

```
fun check(list: List<Int>?) {
    println("The list: \$list")
    println("The list or empty: \$list.orEmpty()")
    println("Is null or empty? " + list.isNullOrEmpty())
}

fun main() {
    check(listOf(1, 2, 3))
    // The list: [1, 2, 3]
    // The list or empty: [1, 2, 3]
    // Is null or empty? false
    check(null)
    // The list: null
    // The list or empty: []
    // Is null or empty? true
    check(listOf())
    // The list: []
    // The list or empty: []
    // Is null or empty? true
}
```

These functions help us operate on nullable values.

null is our friend

Nullability was a source of pain in many languages like Java, where every object can be nullable. As a result, people started avoiding nullability. As a result, you can find suggestions like “Item 43. Return empty arrays or collections, not nulls” from *Effective Java 2nd Edition* by Joshua Bloch. Such practices make literally no sense in Kotlin, where we have a proper nullability system and should not be worried about `null` values. In Kotlin, we treat `null` as our friend, not as a mistake³⁸. Consider the `getUsers` function. There is an essential difference between returning an empty list and `null`. An empty

³⁸See the “Null is your friend, not a mistake” ([link](https://kt.academy/l/re-null) <https://kt.academy/l/re-null>) article by Roman Elizarov, current Project Lead for the Kotlin Programming Language.

list should be interpreted as “the result is an empty list of users because none are available”. The `null` result should be interpreted as “could not produce the result, and the list of users remains unknown”. Forget about outdated practices regarding nullability. The `null` value is our friend in Kotlin³⁹.

lateinit

There are situations where we want to keep a property type not nullable, and yet we cannot specify its value during object creation. Consider properties whose value is injected by a dependency injection framework, or consider properties that are created for every test in the setup phase. Making such properties nullable would lead to inconvenient usage: you would use a not-null assertion even though you know that the value cannot be `null` because it will surely be initialized before usage. For such situations, Kotlin creators introduced the `lateinit` property. Such properties have non-nullable types, and cannot be initialized during creation.

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {

    @Inject
    lateinit var presenter: MainPresenter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        presenter.onCreate()
    }
}

class UserServiceTest {
    lateinit var userRepository: InMemoryUserRepository
    lateinit var userService: UserService
```

³⁹There is more on using nullability in the book *Effective Kotlin*.

```
@Before
fun setup() {
    userRepository = InMemoryUserRepository()
    userService = UserService(userRepository)
}

@Test
fun `should register new user`() {
    // when
    userService.registerUser(aRegisterUserRequest)

    // then
    userRepository.hasUserId(aRegisterUserRequest.id)
    // ...
}
}
```

When we use a `lateinit` property, we must set its value before its first use. If we don't, the program throws an `UninitializedPropertyAccessException` at runtime.

```
lateinit var text: String

fun main() {
    println(text) // RUNTIME ERROR!
    // lateinit property text has not been initialized
}
```

You can always check if a property has been initialized using the `isInitialized` property on its reference. To reference a property, use two colons and a property name⁴⁰.

⁴⁰To reference a property from another object, we need to start with the object before we use `::` and the property name. There is more about referencing properties in the Advanced Kotlin book.

```
lateinit var text: String

private fun printIfInitialized() {
    if (::text.isInitialized) {
        println(text)
    } else {
        println("Not initialized")
    }
}

fun main() {
    printIfInitialized() // Not initialized
    text = "ABC"
    printIfInitialized() // ABC
}
```

Summary

Kotlin offers powerful nullability support that turns nullability from scary and tricky into useful and safe. This is supported by the type system, which separates what is nullable or not nullable. Variables that are nullable must be used safely; for this, we can use safe-calls, not-null assertions, smart-casting, or the Elvis operator. Now, let's finally move on to classes. We've used them many times already, but we finally have everything we need to describe them well.

Classes

Take a look at the world around you and you will likely notice plenty of objects. It might be a book, an Ebook reader, a monitor, or a mug of coffee. We are surrounded by objects. This idea leads to the conclusion that we are living in a world of objects, therefore our programs should be constructed in the same way. This is the conceptual basis of the Object-Oriented Programming approach. Not everyone shares this worldview - some prefer to see the world as a place of possible actions⁴¹, which is the conceptual basis of the Functional Programming approach - but whichever approach we prefer, classes and objects are important structures in Kotlin programming.

A class is a template that is used to create an object with concrete characteristics. To create a class in Kotlin, we use the `class` keyword followed by the name. This is literally all that is needed to create the simplest class because a class body is optional. To create an object, which is an instance of a class, we use the default constructor function, which is the class name and round brackets. Unlike in other languages like C++ or Java, we do not use the `new` keyword in Kotlin.

```
// Simplest class definition
class A

fun main() {
    // Object creation from a class
    val a: A = A()
}
```

Member functions

Inside classes, we can define functions. To do that, we first need to open braces in the class definition in order to specify

⁴¹See “Object-oriented or functional? Two ways to see the world” by Marcin Moskała, link: <https://kt.academy/article/oop-vs-fp>

the class body.

```
class A {  
    // class body  
}
```

There, we can specify functions. Functions defined this way have two important characteristics:

- Functions need to be called on an instance of this class. This means that to call a method, an object needs to be created first.
- Inside methods, we can use `this`, which is a reference to the instance of the class we called this function on.

```
class A {  
    fun printMe() {  
        println(this)  
    }  
}  
  
fun main() {  
    val a = A()  
    println(a) // A@ADDRESS  
    a.printMe() // A@ADDRESS (the same address)  
}
```

All the elements defined inside a class body are called **members**, so a function defined inside a class body is called a **member function**. Functions that are associated with classes are called **methods**, so all member functions are methods, but extension functions (which will be covered in a later chapter) are methods too.

Conceptually speaking, methods represent what an object can do. For instance, a coffee machine should be able to produce coffee, which we might represent by the method `makeCoffee` in the `CoffeeMachine` class. This is how classes with methods help us model the world.

Properties

Inside class bodies, we can also define variables. Variables defined inside classes are called **fields**. There is an important idea known as “encapsulation” which means that fields should never be used directly from outside the class because if that happens, we lose control over their state. Instead, fields should be used through accessors:

- getter - the function that is used to get the current value of this field,
- setter - the function that is used to set new values for this field.

This pattern is highly influential; in Java projects, you can see plenty of getter and setter functions, which are mainly used in classes that hold data. They are needed to achieve encapsulation, but they are also disturbing boilerplate code. So, language creators invented a more powerful concept called “properties”. A **property** is a variable in a class that is automatically encapsulated so that it uses a getter and a setter under the hood. In Kotlin, all variables defined inside classes are properties, not fields.

Some languages, like JavaScript, have built-in support for properties, but Java does not. So, in Kotlin/JVM, accessor functions are generated for each property: a getter for `val`, and a getter and a setter for `var`.

```
// Kotlin code
class User {
    var name: String = ""
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "Alex" // setter call
    println(user.name) // getter call
}
```

```
// equivalent JavaScript code
function User() {
    this.name = '';
}

function main(args) {
    var user = new User();
    user.name = 'Alex';
    println(user.name);
}

// equivalent Java code
public final class User {
    @NotNull
    private String name = "";

    // getter
    @NotNull
    public final String getName() {
        return this.name;
    }

    // setter
    public final void setName(@NotNull String name) {
        this.name = name;
    }
}

public final class PlaygroundKt {
    public static void main(String[] var0) {
        User user = new User();
        user.setName("Alex"); // setter call
        System.out.println(user.getName()); // getter call
    }
}
```

Each property in Kotlin has accessors, therefore we should not define getters or setters using explicit functions. If you

want to change the default accessor, there is a special syntax for that.

```
class User {  
    private var name: String = ""  
  
    // DO NOT DO THAT! DEFINE PROPERTY GETTER INSTEAD  
    fun getName() = name  
  
    // DO NOT DO THAT! DEFINE PROPERTY SETTER INSTEAD  
    fun setName(name: String) {  
        this.name = name  
    }  
}
```

To specify a custom getter, we use the `get` keyword after the property definition. The rest is equivalent to defining a function with no parameters. Inside this function, we use the `field` keyword to reference the backing field. The default getter returns the `field` value, but we can change this behavior so that this value is modified in some way before it is returned. When we define a getter, we can use single-expression syntax or a regular body and the `return` keyword.

```
class User {  
    var name: String = ""  
        get() = field.uppercase()  
    // or  
    // var name: String = ""  
    //     get() {  
    //         return field.uppercase()  
    //     }  
}  
  
fun main() {  
    val user = User()  
    user.name = "norbert"  
    println(user.name) // NORBERT  
}
```

A getter must always have the same visibility and result type as the property. Getters should not throw exceptions and should not perform intensive calculations.

Beware that all property usages are accessors' usages. Inside accessors, you should use `field` instead of the property name because, otherwise, you will likely end up with infinite recursion.

```
class User {  
    // DON'T DO THAT  
    var name: String = ""  
    // Using property name inside getter  
    // leads to infinitive recursion  
    get() = name.uppercase()  
}  
  
fun main() {  
    val user = User()  
    user.name = "norbert"  
    println(user.name) // Error: java.lang.StackOverflowError  
}
```

Setters can be specified similarly, but we need to use the `set` keyword, and we need a single parameter that represents the value that is set. The default setter is used to assign a new value to the `field`, but we can modify this behavior, for instance, to set a new value only if it satisfies some conditions.

```
class User {  
    var name: String = ""  
    get() = field.uppercase()  
    set(value) {  
        if (value.isNotBlank()) {  
            field = value  
        }  
    }  
}
```

```
fun main() {
    val user = User()
    user.name = "norbert"
    user.name = ""
    user.name = " "
    println(user.name) // NORBERT
}
```

Setters might have more restrictive visibility than properties, which we will show in the next chapter.

If a property's custom accessors do not use the `field` keyword, then the backing field will not be generated. For example, we can define a property to represent a full name that is calculated based on a name and surname. This means that some properties might not need a field at all.

```
class User {
    var name: String = ""
    var surname: String = ""
    val fullName: String
        get() = "$name $surname"
}

fun main() {
    val user = User()
    user.name = "Maja"
    user.surname = "Moskała"
    println(user.fullName) // Maja Moskała
}
```

The `fullName` property needs only a getter because it is a read-only `val` property. Whenever we ask for this property's value, a full name will be calculated based on the `name` and `surname`. Notice that this property is calculated on demand, which is an advantage over using a regular property.

```
class User {  
    var name: String = ""  
    var surname: String = ""  
    val fullName1: String  
        get() = "$name $surname"  
    val fullName2: String = "$name $surname"  
}  
  
fun main() {  
    val user = User()  
    user.name = "Maja"  
    user.surname = "Markiewicz"  
    println(user.fullName1) // Maja Markiewicz  
    println(user.fullName2) // Maja Markiewicz  
    user.surname = "Moskała"  
    println(user.fullName1) // Maja Moskała  
    println(user.fullName2) // Maja Markiewicz  
}
```

This difference is only visible when the values our property is based on are mutable; therefore, when we define an immutable object, either calculating the property value on the getter or during class creation should both produce the same result. The difference is in performance: we calculate constant property values during object creation, but getter values are calculated on demand every time they are asked for.

```
class Holder {  
    val v1: Int get() = calculate("v1")  
    val v2: Int = calculate("v2")  
  
    private fun calculate(propertyName: String): Int {  
        println("Calculating $propertyName")  
        return 42  
    }  
}  
  
fun main() {  
    val h1 = Holder() // Calculating v2
```

```
// h1 never used v1, so it was never calculated
// it calculated v2 even though it was not used either
val h2 = Holder() // Calculating v2
println(h2.v1) // Calculating v1 and 42
println(h2.v1) // Calculating v1 and 42
println(h2.v2) // 42
println(h2.v2) // 42
// h2 used v1 two times, and it was calculated two times
// it calculated v2 only once,
// even though it was used two times
}
```

As another example, let's imagine we need to keep the user's birthdate. Initially, we represented it with `Date` from the Java Standard Library.

```
import java.util.Date

class User {
    // ...
    var birthdate: Date? = null
}
```

Time has passed, and `Date` is no longer a good way to represent this attribute. Maybe we have problems with serialization; maybe we need to make our object multiplatform; or maybe we need to represent time in another calendar not supported by `Date`. So, we've decided to use a different type instead of `Date`. Let's say that we've decided to use a `Long` property to keep milliseconds, but we cannot get rid of the previous property because it is used by many other parts of our code. To have our cake and eat it, we can transform our `birthdate` property to fully depend on the new representation. This way, we have changed how the birthdate is represented without changing the previous usage.

```
class User {  
    // ...  
    var birthdateMillis: Long? = null  
  
    var birthdate: Date?  
        get() = birthdateMillis?.let(::Date)  
        set(value) {  
            birthdateMillis = value?.time  
        }  
}
```

In the above getter, I use `let` and a constructor reference. Both these Kotlin features are explained in the book **Functional Kotlin**.

Such a `birthdate` property can also be defined as an extension function, which will be presented in the chapter **Extensions**.

Constructors

When we create an object, we often want to initialize it with specific values. This is what we use constructors for. As we've seen already, when no constructors are specified, an empty default constructor is generated with no parameters.

```
class A  
  
val a = A()
```

To specify our custom constructor, the classic way is to use the `constructor` keyword inside the class body, and then we define its parameters and body.

```
class User {  
    var name: String = ""  
    var surname: String = ""  
  
    constructor(name: String, surname: String) {  
        this.name = name  
        this.surname = surname  
    }  
}  
  
fun main() {  
    val user = User("Johnny", "Depp")  
    println(user.name) // Johnny  
    println(user.surname) // Depp  
}
```

Constructors are typically used to set initial values of our properties. To simplify this, Kotlin introduced a special kind of constructor called the **primary constructor**. It is defined just after the class name, and its parameters can be used during the initialization of properties.

```
class User constructor(name: String, surname: String) {  
    var name: String = name  
    var surname: String = surname  
}  
  
fun main() {  
    val user = User("Johnny", "Depp")  
    println(user.name) // Johnny  
    println(user.surname) // Depp  
}
```

When we specify a primary constructor, use of the `constructor` keyword is optional, so we can just skip it.

```
class User(name: String, surname: String) {  
    var name: String = name  
    var surname: String = surname  
}  
  
fun main() {  
    val user = User("Johnny", "Depp")  
    println(user.name) // Johnny  
    println(user.surname) // Depp  
}
```

There can be only one primary constructor. We can define another constructor, called **secondary constructor**, but it needs to call the primary constructor using the `this` keyword.

```
class User(name: String, surname: String) {  
    var name: String = name  
    var surname: String = surname  
  
    // Secondary constructor  
    constructor(user: User) : this(user.name, user.surname) {  
        // optional body  
    }  
}  
  
fun main() {  
    val user = User("Johnny", "Depp")  
    println(user.name) // Johnny  
    println(user.surname) // Depp  
  
    val user2 = User(user)  
    println(user2.name) // Johnny  
    println(user2.surname) // Depp  
}
```

The primary constructor is typically used to specify initial values for our properties. These properties often have the same names as the parameters, so Kotlin introduced better

support for this: we can define properties inside the primary constructor. Such properties define a class property and a constructor parameter, both of which have the same name.

```
class User(  
    var name: String,  
    var surname: String,  
) {  
    // optional body  
}  
  
fun main() {  
    val user = User("Johnny", "Depp")  
    println(user.name) // Johnny  
    println(user.surname) // Depp  
}
```

This is how the vast majority of Kotlin classes are defined: using a primary constructor with properties. We rarely use other kinds of constructors.

We often define primary constructors with default values. Here, we create an instance of `User` without providing the `surname` argument, so the default value we specified will be used during `User` creation.

```
class User(  
    var name: String = "",  
    var surname: String = "Anonim",  
)  
  
fun main() {  
    val user = User("Johnny")  
    println(user.name) // Johnny  
    println(user.surname) // Anonim  
}
```

Classes representing data in Kotlin and Java

When comparing classes defined in Kotlin and Java, we can see how much boilerplate code Kotlin has eliminated. In Java, to represent a simple user, with a name, surname, and age, the typical implementation looks as follows:

```
public final class User {  
    @NotNull  
    private final String name;  
    @NotNull  
    private final String surname;  
    private final int age;  
  
    public User(  
        @NotNull String name,  
        @NotNull String surname,  
        int age  
    ) {  
        this.name = name;  
        this.surname = surname;  
        this.age = age;  
    }  
  
    @NotNull  
    public String getName() {  
        return name;  
    }  
  
    @NotNull  
    public String getSurname() {  
        return surname;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

In Kotlin, we represent the same class in the following way:

```
class User(  
    val name: String,  
    val surname: String,  
    val age: Int?,  
)
```

The result of the compilation is practically the same. Getters and constructors are there. If you don't believe it, check it yourself (as presented in the *What is under the hood on JVM?* section in the *Your first program in Kotlin* chapter). Kotlin is a concise but powerful language.

Inner classes

In Kotlin, we can define classes inside classes. They are static by default, which means that they do not have access to outer classes, therefore they can be created without a reference to an outer class.

```
class Puppy(val name: String) {  
  
    class InnerPuppy {  
        fun think() {  
            // we have no access to name here  
            println("Inner puppy is thinking")  
        }  
    }  
}  
  
fun main() {  
    val innerPuppy = Puppy.InnerPuppy()  
    // We create InnerPuppy on class, not object  
    innerPuppy.think() // Inner puppy is thinking  
}
```

If you want your inner class to have a reference to its outer class, you need to make it inner using the `inner` modifier. However, creating objects from such classes requires an instance of the outer class.

```
class Puppy(val name: String) {

    inner class InnerPuppy {
        fun think() {
            println("Inner $name is thinking")
        }
    }

    fun main() {
        val puppy = Puppy("Cookie")
        val innerPuppy = puppy.InnerPuppy() // We need puppy
        innerPuppy.think() // Inner Cookie is thinking
    }
}
```

Examples of inner classes in the standard library are:

- private implementations of iterators;
- classes, where there is a close association between the outer class and the inner class, and the inner class is used to not define another name in the library namespace.

```
// A class from Kotlin stdlib
class FileTreeWalk(
    ...
) : Sequence<File> {

    /** Returns an iterator walking through files. */
    override fun iterator(): Iterator<File> =
        FileTreeWalkIterator()

    private inner class FileTreeWalkIterator
```

```
: AbstractIterator<File>() {  
    ...  
}  
  
...  
}
```

Summary

As you can see, in Kotlin we can define classes using really concise syntax, and the result is very readable. The primary constructor is an amazing invention, as is the fact that Kotlin uses properties instead of fields. You have also learned about inner classes. This is all great, but we haven't yet touched on inheritance, which is so important for developers who like Object-Oriented style. We will discuss this along with interfaces and abstract classes in the next chapter.

Inheritance

Ancient philosophers observed that many classes of objects share the same characteristics⁴². For instance, all mammals have hair or fur, are warm-blooded, and feed their young with milk⁴³. In programming, we represent such relationships using **inheritance**.

When a class inherits from another class, it has all its member functions and properties. The class that inherits is known as a **subclass** of the class it inherits from, which is called a **superclass**. They are also known as **child** and **parent**.

In Kotlin, all classes are closed by default, which means we cannot inherit from them. We need to open a class using the `open` keyword to allow inheritance from it. To inherit from a class, we place a colon after the primary constructor (or after the class name if there is no primary constructor), and then we invoke the superclass constructor. In the example below, the class `Dog` inherits from the class `Mammal`. Since `Mammal` has no constructor specified, we call it without arguments, so with `Mammal()`. This way, `Dog` inherits all the properties and methods from `Mammal`.

```
open class Mammal {  
    val haveHairOrFur = true  
    val warmBlooded = true  
    var canFeed = false  
  
    fun feedYoung() {  
        if (canFeed) {  
            println("Feeding young with milk")  
        }  
    }  
}
```

⁴²I believe it was Aristotle who mainly contributed to the development and popularization of this idea.

⁴³A fun fact for dog or cat owners is that, like all mammals, they too have belly buttons; however, it is often not easy to find them as they are small and sometimes hidden under fur.

```
}
```

```
class Dog : Mammal() {
    fun makeVoice() {
        println("Bark bark")
    }
}

fun main() {
    val dog = Dog()
    dog.makeVoice() // Bark bark
    println(dog.haveHairOrFur) // true
    println(dog.warmBlooded) // true
    // Dog is Mammal, so we can up-cast it
    val mammal: Mammal = dog
    mammal.canFeed = true
    mammal.feedYoung() // Feeding young with milk
}
```

Conceptually, we treat subclasses as if they *are* their superclasses: so, if `Dog` inherits from `Mammal`, we say that `Dog` **is** a `Mammal`. Likewise, wherever `Mammal` is expected, we can use an instance of `Dog`. Taking this into account, inheritance should only be used if a real “*is a*” relationship between two classes exists.

Overriding elements

By default, subclasses cannot override elements defined in superclasses. To make this possible, these elements need to allow it with the `open` modifier, because in Kotlin all elements are closed by default. Then, subclasses can override their parents’ implementation, which looks just like defining the same function in children but with the `override` modifier (this modifier is required in Kotlin).

```
open class Mammal {
    val haveHairOrFur = true
    val warmBlooded = true
    var canFeed = false

    open fun feedYoung() {
        if (canFeed) {
            println("Feeding young with milk")
        }
    }
}

class Cat : Mammal() {
    override fun feedYoung() {
        if (canFeed) {
            println("Feeding young with milk")
        } else {
            println("Feeding young with milk from bottle")
        }
    }
}

fun main() {
    val dog = Mammal()
    dog.feedYoung() // Nothing printed
    val cat = Cat()
    cat.feedYoung() // Feeding young with milk from bottle
    cat.canFeed = true
    cat.feedYoung() // Feeding young with milk
}
```

Parents with non-empty constructors

So far, we have inherited only from classes with empty constructors; so, when we were specifying the superclass, we used empty parentheses. However, if the superclass has constructor parameters, we need to define some arguments inside these parentheses.

```
open class Animal(val name: String)

class Dodo : Animal("Dodo")
```

We can use primary constructor properties as superclass constructor arguments or construct these arguments.

```
open class Animal(val name: String)

class Dog(name: String) : Animal(name)

class Cat(name: String) : Animal("Mr $name")

class Human(
    firstName: String,
    lastName: String,
) : Animal("$firstName $lastName")

fun main() {
    val dog = Dog("Cookie")
    println(dog.name) // Cookie
    val cat = Cat("MiauMiau")
    println(cat.name) // Mr MiauMiau
}
```

Super call

When a class extends another class, it takes the behavior from the superclass but also adds some behavior that is specific to the subclass. This is why overriding methods often need to include the behavior of the methods they override. For this, it is useful to call the superclass implementation in these subclass methods. We do this using the `super` keyword and a dot and then call the method we override.

Consider the classes `Dog` and `BorderCollie` that are presented in the example below. The default behavior for a dog is to wave its tail when it sees a dog friend. Border Collies should behave

the same but additionally lie down. In this case, to call the superclass implementation, we need to use `super.seeFriend()`.

```
open class Dog {
    open fun seeFriend() {
        println("Wave its tail")
    }
}

class BorderCollie : Dog() {
    override fun seeFriend() {
        println("Lie down")
        super.seeFriend()
    }
}

fun main() {
    val dog = Dog()
    dog.seeFriend() // Wave its tail
    val borderCollie = BorderCollie()
    borderCollie.seeFriend()
    // Lie down
    // Wave its tail
}
```

Abstract class

A mammal is a group of animals, not a concrete species. It defines a set of characteristics but might not exist in itself. To define a class that can only be used as a subclass of other classes but cannot produce an object, we use the `abstract` keyword before its class definition. You can interpret the `open` modifier as “one can inherit from this class”, whereas `abstract` means “one must inherit from this class to use it”.

```
abstract class Mammal {  
    val haveHairOrFur = true  
    val warmBlooded = true  
    var canFeed = false  
  
    fun feedYoung() {  
        if (canFeed) {  
            println("Feeding young with milk")  
        }  
    }  
}
```

Abstract classes are open, so there is no need to use the `open` modifier when a class has the `abstract` modifier already.

When a class is abstract, it can have abstract functions and properties. Such functions do not have a body, and each subclass needs to override them. Thanks to that, when we have an object whose type is an abstract class, we can call its abstract functions because whatever the actual class of this object is, it still needs to define these functions.

```
abstract class Mammal {  
    val haveHairOrFur = true  
    val warmBlooded = true  
    var canFeed = false  
  
    abstract fun feedYoung()  
}  
  
class Dog : Mammal() {  
    override fun feedYoung() {  
        if (canFeed) {  
            println("Feeding young with milk")  
        }  
    }  
}  
  
class Human : Mammal() {
```

```
override fun feedYoung() {
    if (canFeed) {
        println("Feeding young with milk")
    } else {
        println("Feeding young with milk from bottle")
    }
}

fun feedYoung(mammal: Mammal) {
    // We can do that, because feedYoung is an abstract
    // function in Mammal
    mammal.feedYoung()
}

fun main() {
    val dog = Dog()
    dog.canFeed = true
    feedYoung(dog) // Feeding young with milk
    feedYoung(Human()) // Feeding young with milk from bottle
}
```

An abstract class can also have non-abstract methods, which have a body. Such methods can be used by other methods. Therefore, abstract classes can be used as templates with partial implementation for other classes. Consider the `CoffeeMachine` abstract class below, which specifies how latte or doppio can be prepared, but it needs a subclass to override the `prepareEspresso` and `addMilk` methods. This class provides implementation for only some methods, so it is a partial implementation.

```
abstract class CoffeeMachine {  
    abstract fun prepareEspresso()  
    abstract fun addMilk()  
  
    fun prepareLatte() {  
        prepareEspresso()  
        addMilk()  
    }  
    fun prepareDoppio() {  
        prepareEspresso()  
        prepareEspresso()  
    }  
}
```

Kotlin does not support multiple inheritance, so a class can inherit only from one open class. I do not find this a problem because inheritance is not so popular nowadays - interfaces are implemented instead.

Interfaces

An interface defines a set of properties and methods that a class should have. We define interfaces with the `interface` keyword, a name, and a body with the expected properties and methods.

```
interface CoffeeMaker {  
    val type: String  
    fun makeCoffee(size: Size): Coffee  
}
```

When a class implements an interface, this class has to override all the elements defined by this interface. Thanks to that, we can treat an instance of a class as an instance of an interface. We implement interfaces similarly to how we extend classes, but without calling a constructor because interfaces cannot have constructors.

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

interface UserRepository {
    fun findUser(id: Int): User?
    fun addUser(user: User)
}

class FakeUserRepository : UserRepository {
    private var users = mapOf<Int, User>()

    override fun findUser(id: Int): User? = users[id]

    override fun addUser(user: User) {
        users += user.id to user
    }
}

fun main() {
    val repo: UserRepository = FakeUserRepository()
    repo.addUser(User(123, "Zed"))
    val user = repo.findUser(123)
    println(user?.name) // Zed
}
```

Interfaces can specify default bodies for their methods. Such methods do not need to (but can) be implemented by subclasses.

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

interface UserRepository {
    fun findUser(id: Int): User? =
        getUsers().find { it.id == id }

    fun getUsers(): List<User>
}

class FakeUserRepository : UserRepository {
    private var users = listOf<User>()
```

```
override fun getUsers(): List<User> = users

fun addUser(user: User) {
    users += user
}

fun main() {
    val repo = FakeUserRepository()
    repo.addUser(User(123, "Zed"))
    val user = repo.findUser(123)
    println(user?.name) // Zed
}
```

As mentioned already, interfaces can specify that they expect a class to have a particular property. Such properties can either be defined as regular properties, or they can be defined by accessors (getter for `val`, or getter and setter for `var`).

```
interface Named {
    val name: String
    val fullName: String
}

class User(
    override val name: String,
    val surname: String,
) : Named {
    override val fullName: String
        get() = "$name $surname"
}
```

The read-only `val` property can be overridden with a read-write `var` property. This is because the `val` property expects a getter, and the `var` property provides a getter as well as an additional setter.

```
interface Named {  
    val name: String  
}  
  
class NameBox : Named {  
    override var name = "(default)"  
}
```

A class can implement multiple interfaces.

```
interface Drinkable {  
    fun drink()  
}  
  
interface Spillable {  
    fun spill()  
}  
  
class Mug : Drinkable, Spillable {  
    override fun drink() {  
        println("Ummm")  
    }  
    override fun spill() {  
        println("Ow, ow, OWWW")  
    }  
}
```

We can specify a body for a method in an interface. Such a body is treated as the default body and will be used by the

classes implementing this interface⁴⁴. We can call this default body using the `super` keyword and a regular method call.

```
interface NicePerson {
    fun cheer() {
        println("Hello")
    }
}

class Alex : NicePerson

class Ben : NicePerson {
    override fun cheer() {
        super.cheer()
        println("My name is Ben")
    }
}

fun main() {
    val alex = Alex()
    alex.cheer() // Hello

    val ben = Ben()
    ben.cheer()
```

⁴⁴Default methods make interfaces more than what they were traditionally considered to be. They make it possible for interfaces to define behavior that is inherited by classes that implement these interfaces. The concept which represents a set of methods that can be used to extend the functionality of a class is known in programming as a **trait**. This is why in early versions of Kotlin, we used the `trait` keyword instead of the `interface` keyword. However, version 8 of Java introduced default bodies for interface methods, so Kotlin creators assumed that the JVM community would expand the meaning of an interface, and this is why we now use the `interface` keyword. The concept of traits is used in Kotlin. An example can be found in my article *Traits for testing in Kotlin*, which you can find under <http://kt.academy/article/traits-testing>.

```
// Hello  
// My name is Ben  
}
```

When two interfaces define a method with the same name and parameters, the class that implements both these interfaces **must** override this method. To call the default bodies of these methods, we need to use `super` with the name of the class we want to use in angle brackets. So, to call `start` from `Boat` use `super<Boat>.start()`. Or, to call `start` from `Car` use `super<Car>.start()`.

```
interface Boat {  
    fun start() {  
        println("Ready to swim")  
    }  
}  
  
interface Car {  
    fun start() {  
        println("Ready to drive")  
    }  
}  
  
class Amphibian: Car, Boat {  
    override fun start() {  
        super<Car>.start()  
        super<Boat>.start()  
    }  
}  
  
fun main() {  
    val vehicle = Amphibian()  
    vehicle.start()  
    // Ready to drive  
    // Ready to swim  
}
```

Visibility

When we design our classes, we prefer to expose as little as possible⁴⁵. If there is no reason for an element to be visible⁴⁶, we prefer to keep it hidden. This is why if there is no good reason to have a less restrictive visibility type, it is good practice to make the visibility of classes and elements as restrictive as possible. We do this using visibility modifiers.

For class members, these are the 4 visibility modifiers we can use:

- `public` (default) - visible everywhere for clients which can see the declaring class.
- `private` - visible inside this class only.
- `protected` - visible inside this class and in subclasses.
- `internal` - visible inside this module for clients which can see the declaring class.

Top-level elements have 3 visibility modifiers:

- `public` (default) - visible everywhere.
- `private` - visible inside the same file only.
- `internal` - visible inside this module.

Note that a module is not the same as a package. In Kotlin, a module is defined as a set of Kotlin sources that are compiled together. This might mean:

- a Gradle source set,
- a Maven project,

⁴⁵A deeper explanation of the reasons behind this general programming rule is presented in Effective Kotlin, Item 30: Minimize elements' visibility.

⁴⁶Visibility defines where an element can be used. If an element is not visible, it will not be suggested by the IDE and cannot be used.

- an IntelliJ IDEA module,
- a set of files compiled with one invocation of an Ant task.

Let's see some examples, starting with the default visibility, which makes elements visible everywhere and can be explicitly specified using the `public` modifier.

```
// File1.kt
open class A {
    public val a = 10
    public fun b() {
        println(a) // Can use it
    }
}

public val c = 20
public fun d() {}

class B: A() {
    fun e() {
        println(a) // Can use it
        println(b()) // Can use it
    }
}

fun main() {
    println(A().a) // Can use it
    println(A().b()) // Can use it
    println(c) // Can use it
    println(d()) // Can use it
}

// File2.kt in the same or different module as File1.kt
fun main() {
    println(A().a) // Can use it
    println(A().b()) // Can use it
    println(c) // Can use it
    println(d()) // Can use it
}
```

The `private` modifier can be interpreted as “visible in the creation scope”; so, if we define an element in a class, it will be visible only in this class; if we define an element in a file, it will be visible only in this file.

```
// File1.kt
open class A {
    private val a = 10
    private fun b() {
        println(a) // Can use it
    }
}

private val c = 20
private fun d() {}

class B : A() {
    fun e() {
        println(a) // Error, cannot use a!
        println(b()) // Error, cannot use b!
    }
}

fun main() {
    println(A().a) // Error, cannot use a!
    println(A().b()) // Error, cannot use b!
    println(c) // Can use it
    println(d()) // Can use it
}

// File2.kt in the same or different module as File1.kt
fun main() {
    println(A().a) // Error, cannot use a!
    println(A().b()) // Error, cannot use b!
    println(c) // Error, cannot use c!
    println(d()) // Error, cannot use d!
}
```

The `protected` modifier can be interpreted as “visible in the

class and its subclasses”. `protected` only makes sense for elements defined inside classes. It is similar to `private`, but `protected` elements are also visible inside subclasses of the class where these elements are defined.

```
// File1.kt
open class A {
    protected val a = 10
    protected fun b() {
        println(a) // Can use it
    }
}

open class B: A() {
    fun e() {
        println(a) // Can use it!
        println(b()) // Can use it!
    }
}

class C: A() {
    fun f() {
        println(a) // Can use it!
        println(b()) // Can use it!
    }
}

fun main() {
    println(A().a) // Error, cannot use a!
    println(A().b()) // Error, cannot use b!
}

// File2.kt in the same or different module as File1.kt
fun main() {
    println(A().a) // Error, cannot use a!
    println(A().b()) // Error, cannot use b!
}
```

The `internal` modifier makes elements visible in the same

module. It is useful for library creators who use the `internal` modifier for elements they want to be visible in their project but don't want to expose to library users. It is also useful in multi-module projects to limit access to a single module. It's useless in single-module projects⁴⁷.

```
// File1.kt
open class A {
    internal val a = 10
    internal fun b() {
        println(a) // Can use it
    }
}

internal val c = 20
internal fun d() {}

class B: A() {
    fun e() {
        println(a) // Can use it
        println(b()) // Can use it
    }
}

fun main() {
    println(A().a) // Can use it
    println(A().b()) // Can use it
    println(c) // Can use it
    println(d()) // Can use it
}
```

⁴⁷However, I've seen cases where teams used the `internal` visibility modifier as a substitute for the Java package-private modifier. Even though it has different behavior, some developers treat this modifier as a form of documentation that should be interpreted as "this element should not be used in different packages". I am not a fan of such practices, therefore I suggest using annotation instead.

```
// File2.kt in the same module as File1.kt
fun main() {
    println(A().a) // Can use it
    println(A().b()) // Can use it
    println(c) // Can use it
    println(d()) // Can use it
}

// File3.kt in a different module than File1.kt
fun main() {
    println(A().a) // Error, cannot use a!
    println(A().b()) // Error, cannot use b!
    println(c) // Error, cannot use c!
    println(d()) // Error, cannot use d!
}
```

If your module might be used by another module, change the visibility of public elements that you don't want to expose to `internal`. If an element is designed for inheritance and is only used in a class and subclasses, make it `protected`. If you use an element only in the same file or class, make it `private`.

Changing the visibility of a property means changing the visibility of its accessors. A property's field is always private. To change setter visibility, place the visibility modifier before the `set` keyword. The getter must have the same visibility as the property.

```
class View {
    var isVisible: Boolean = true
        private set

    fun hide() {
        isVisible = false
    }
}

fun main() {
    val view = View()
```

```
    println(view.isVisible) // true
    view.hide()
    println(view.isVisible) // false
    view.isVisible = true // ERROR
    // Cannot assign to 'isVisible',
    // the setter is private in 'View'
}
```

Any

If a class has no explicit parent, its implicit parent is `Any`, which is a superclass of all the classes in Kotlin. This means that when we expect the `Any?` type parameter, we accept all possible objects as arguments.

```
fun consumeAnything(a: Any?) {
    println("Om nom $a")
}

fun main() {
    consumeAnything(null) // Om nom null
    consumeAnything(123) // Om nom 123
    consumeAnything("ABC") // Om nom ABC
}
```

You can think of `Any` as an open class with three methods: `toString`, `equals` and `hashCode`. These will be better explained in the next chapter, `Data classes`. Overriding methods defined by `Any` is optional because each is an open function with a default body.

Summary

In this chapter, we've learned how to use inheritance in Kotlin. We've got familiar with open and abstract classes, interfaces, and visibility modifiers. These are useful when we want to represent hierarchies of classes.

Instead of using classes to represent hierarchies, we can also treat them as holders of data; for this we use the `data` modifier, which is presented in the next chapter.

Data classes

In Kotlin, we say that all classes inherit from the `Any` super-class, which is at the top of the class hierarchy⁴⁸. Methods defined in `Any` can be called on all objects. These methods are:

- `equals` - used when two objects are compared using `==`,
- `hashCode` - used by collections that use the hash table algorithm,
- `toString` - used to represent an object as a string, e.g., in a string template or the `print` function.

Thanks to these methods, we can represent any object as a string or check the equality of any two objects.

```
// Any formal definition
open class Any {
    open operator fun equals(other: Any?): Boolean
    open fun hashCode(): Int
    open fun toString(): String
}

class A // Implicitly inherits from Any

fun main() {
    val a = A()
    a.equals(a)
    a == a
    a.hashCode()
    a.toString()
    println(a)
}
```

Truth be told, `Any` is represented as a class, but it should actually be considered the head of the

⁴⁸So `Any` is an analog to `Object` in Java, JavaScript or C#. There is no direct analog in C++.

type hierarchy, but with some special functions. Consider the fact that `Any` is also the supertype of all interfaces, even though interfaces cannot inherit from classes.

The default implementations of `equals`, `hashCode`, and `toString` are strongly based on the object's address in memory. The `equals` method returns `true` only when the address of both objects is the same, which means the same object is on both sides. The `hashCode` method typically transforms an address into a number. `toString` produces a string that starts with the class name, then the at sign “@”, then the unsigned hexadecimal representation of the hash code of the object.

```
class A

fun main() {
    val a1 = A()
    val a2 = A()

    println(a1.equals(a1)) // true
    println(a1.equals(a2)) // false
    // or
    println(a1 == a1) // true
    println(a1 == a2) // false

    println(a1.hashCode()) // Example: 149928006
    println(a2.hashCode()) // Example: 713338599

    println(a1.toString()) // Example: A@8efb846
    println(a2.toString()) // Example: A@2a84aee7
    // or
    println(a1) // Example: A@8efb846
    println(a2) // Example: A@2a84aee7
}
```

By overriding these methods, we can decide how a class should behave. Consider the following class `A`, which is equal

to other instances of the same class and returns a constant hash code and string representation.

```
class A {  
    override fun equals(other: Any?): Boolean = other is A  
  
    override fun hashCode(): Int = 123  
  
    override fun toString(): String = "A()"  
}  
  
fun main() {  
    val a1 = A()  
    val a2 = A()  
  
    println(a1.equals(a1)) // true  
    println(a1.equals(a2)) // true  
    // or  
    println(a1 == a1) // true  
    println(a1 == a2) // true  
  
    println(a1.hashCode()) // 123  
    println(a2.hashCode()) // 123  
  
    println(a1.toString()) // A()  
    println(a2.toString()) // A()  
    // or  
    println(a1) // A()  
    println(a2) // A()  
}
```

I've dedicated separate items in the Effective Kotlin book to implementing a custom `equals` and `hashCode`⁴⁹, but in practice we rarely need to do that. As it turns out, in modern projects we almost solely operate on only two kinds of objects:

⁴⁹These are Item 42: Respect the contract of `equals` and Item 43: Respect the contract of `hashCode`.

- Active objects, like services, controllers, repositories, etc. Such classes don't need to override any methods from `Any` because the default behavior is perfect for them.
- Data model class objects, which represent bundles of data. For such objects, we use the `data` modifier, which overrides the `toString`, `equals`, and `hashCode` methods. The `data` modifier also implements the methods `copy` and `componentN` (`component1`, `component2`, etc.), which are not inherited and cannot be modified⁵⁰.

```
data class Player(  
    val id: Int,  
    val name: String,  
    val points: Int  
)  
  
val player = Player(0, "Gecko", 9999)
```

Let's discuss the aforementioned implicit data class methods and the differences between regular class behavior and data class behavior.

Transforming to a string

The default `toString` transformation produces a string that starts with the class name, then the at sign “@”, and then the unsigned hexadecimal representation of the hash code of the object. The purpose of this is to display the class name and to determine whether two strings represent the same object or not.

⁵⁰This type of class is so popular that in Java it is common practice to auto-generate `equals`, `hashCode`, and `toString` in IntelliJ or using the Lombok library.

```
class FakeUserRepository

fun main() {
    val repository1 = FakeUserRepository()
    val repository2 = FakeUserRepository()
    println(repository1) // e.g. FakeUserRepository@8efb846
    println(repository1) // e.g. FakeUserRepository@8efb846
    println(repository2) // e.g. FakeUserRepository@2a84aee7
}
```

With the `data` modifier, the compiler generates a `toString` that displays the class name and then pairs with the name and value for each primary constructor property. We assume that data classes are represented by their primary constructor properties, so all these properties, together with their values, are displayed during a transformation to a string. This is useful for logging and debugging.

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

fun main() {
    val player = Player(0, "Gecko", 9999)
    println(player)
    // Player(id=0, name=Gecko, points=9999)
    println("Player: $player")
    // Player: Player(id=0, name=Gecko, points=9999)
}
```



Objects equality

In Kotlin, we check the equality of two objects using `==`, which uses the `equals` method from `Any`. So, this method decides if two objects should be considered equal or not. By default, two different instances are never equal. This is perfect for active objects, i.e., objects that work independently of other instances of the same class and possibly have a protected mutable state.

```
class FakeUserRepository

fun main() {
    val repository1 = FakeUserRepository()
    val repository2 = FakeUserRepository()
    println(repository1 == repository1) // true
    println(repository1 == repository2) // false
}
```

Classes with the `data` modifier represent bundles of data; they are considered equal to other instances if:

- both are of the same class,
- their primary constructor property values are equal.

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

fun main() {
    val player = Player(0, "Gecko", 9999)
    println(player == Player(0, "Gecko", 9999)) // true
    println(player == Player(0, "Ross", 9999)) // false
}
```

This is what a simplified implementation of the `equals` method generated by the `data` modifier for the `Player` class looks like:

```
override fun equals(other: Any?): Boolean = other is Player &&
    other.id == this.id &&
    other.name == this.name &&
    other.points == this.points
```

Implementing a custom `equals` is described in Effective Kotlin, Item 42: Respect the contract of `equals`.

Hash code

Another method from `Any` is `hashCode`, which is used to transform an object into an `Int`. With a `hashCode` method, the object instance can be stored in the hash table data structure implementations that are part of many popular classes, including `HashSet` and `HashMap`. The most important rule of the `hashCode` implementation is that it should:

- be consistent with `equals`, so it should return the same `Int` for equal objects, and it must always return the same hash code for the same object.
- spread objects as uniformly as possible in the range of all possible `Int` values.

The default `hashCode` is based on an object's address in memory. The `hashCode` generated by the `data` modifier is based on the hash codes of this object's primary constructor properties. In both cases, the same number is returned for equal objects.

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

fun main() {
    println(Player(0, "Gecko", 9999).hashCode()) // 2129010918
    println(Player(0, "Gecko", 9999).hashCode()) // 2129010918
    println(Player(0, "Ross", 9999).hashCode()) // 79159602
}
```

To learn more about the hash table algorithm and implementing a custom `hashCode` method, see *Effective Kotlin, Item 41: Respect the contract of hashCode*.

Copying objects

Another method generated by the `data` modifier is `copy`, which is used to create a new instance of a class but with a concrete modification. The idea is very simple: it is a function with parameters for each primary constructor property, but each of these parameters has a default value, i.e., the current value of the associated property.

```
// This is how copy generated by data modifier
// for Person class looks like under the hood
fun copy(
    id: Int = this.id,
    name: String = this.name,
    points: Int = this.points
) = Player(id, name, points)
```

This means we can call `copy` with no parameters to make a copy of our object with no modifications, but we can also specify new values for the properties we want to change.

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

fun main() {
    val p = Player(0, "Gecko", 9999)

    println(p.copy()) // Player(id=0, name=Gecko, points=9999)

    println(p.copy(id = 1, name = "New name"))
    // Player(id=1, name=New name, points=9999)
```

```
    println(p.copy(points = p.points + 1))
    // Player(id=0, name=Gecko, points=10000)
}
```

Note that `copy` creates a shallow copy of an object; so, if our object holds a mutable state, a change in one object will be a change in all its copies too.

```
data class StudentGrades(
    val studentId: String,
    // Code smell: Avoid using mutable objects in data classes
    val grades: MutableList<Int>
)

fun main() {
    val grades1 = StudentGrades("1", mutableListOf())
    val grades2 = grades1.copy(studentId = "2")
    println(grades1) // Grades(studentId=1, grades=[])
    println(grades2) // Grades(studentId=2, grades=[])
    grades1.grades.add(5)
    println(grades1) // Grades(studentId=1, grades=[5])
    println(grades2) // Grades(studentId=2, grades=[5])
    grades2.grades.add(1)
    println(grades1) // Grades(studentId=1, grades=[5, 1])
    println(grades2) // Grades(studentId=2, grades=[5, 1])
}
```

We do not have this problem when we use `copy` for immutable classes, i.e., classes with only `val` properties that hold immutable values. `copy` was introduced as special support for immutability (for details, see *Effective Kotlin, Item 1: Limit mutability*).

```
data class StudentGrades(  
    val studentId: String,  
    val grades: List<Int>  
)  
  
fun main() {  
    var grades1 = StudentGrades("1", listOf())  
    var grades2 = grades1.copy(studentId = "2")  
    println(grades1) // Grades(studentId=1, grades=[])  
    println(grades2) // Grades(studentId=2, grades=[])  
    grades1 = grades1.copy(grades = grades1.grades + 5)  
    println(grades1) // Grades(studentId=1, grades=[5])  
    println(grades2) // Grades(studentId=2, grades=[])  
    grades2 = grades2.copy(grades = grades2.grades + 1)  
    println(grades1) // Grades(studentId=1, grades=[5])  
    println(grades2) // Grades(studentId=2, grades=[1])  
}
```

Notice that data classes are unsuitable for objects that must maintain invariant constraints on mutable properties. For example, in the `User` example below, the class would not be able to guarantee that the `name` and `surname` values are not blank if these variables were mutable (so, defined with `var`). Data classes are perfectly fit for immutable properties, whose constraints might be checked during the creation of these objects. In the example below, we can be sure that the `name` and `surname` values are not blank in an instance of `User`.

```
data class User(  
    val name: String,  
    val surname: String,  
) {  
    init {  
        require(name.isNotBlank())  
        // throws exception if name is blank  
        require(surname.isNotBlank())  
        // throws exception if surname is blank  
    }  
}
```

Destructuring

Kotlin supports a feature called position-based destructuring, which lets us assign multiple variables to components of a single object. For that, we place our variable names in round brackets.

```
data class Player(  
    val id: Int,  
    val name: String,  
    val points: Int  
)  
  
fun main() {  
    val player = Player(0, "Gecko", 9999)  
    val (id, name, pts) = player  
    println(id) // 0  
    println(name) // Gecko  
    println(pts) // 9999  
}
```

This mechanism relies on position, not names. The object on the right side of the equality sign needs to provide the functions `component1`, `component2`, etc., and the variables are assigned to the results of these methods.

```
val (id, name, pts) = player  
// is compiled to  
val id: Int = player.component1()  
val name: String = player.component2()  
val pts: Int = player.component3()
```

This code works because the `data` modifier generates `componentN` functions for each primary constructor parameter, according to their order in the constructor.

These are currently all the functionalities that the `data` modifier provides. Don't use it if you don't need `toString`, `equals`,

`hashCode`, `copy` or destructuring. If you need some of these functionalities for a class representing a bundle of data, use the `data` modifier instead of implementing the methods yourself.

When and how should we use destructuring?

Position-based destructuring has pros and cons. Its biggest advantage is that we can name variables however we want, so we can use names like `country` and `city` in the example below. We can also destructure anything we want as long as it provides `componentN` functions. This includes `List` and `Map.Entry`, both of which have `componentN` functions defined as extensions:

```
fun main() {
    val visited = listOf("Spain", "Morocco", "India")
    val (first, second, third) = visited
    println("$first $second $third")
    // Spain Morocco India

    val trip = mapOf(
        "Spain" to "Gran Canaria",
        "Morocco" to "Taghazout",
        "India" to "Rishikesh"
    )
    for ((country, city) in trip) {
        println("We loved $city in $country")
        // We loved Gran Canaria in Spain
        // We loved Taghazout in Morocco
        // We loved Rishikesh in India
    }
}
```

On the other hand, position-based destructuring is dangerous. We need to adjust every destructuring when the order or number of elements in a data class changes. When we use this feature, it is very easy to introduce errors into our code by changing the order of the primary constructor's properties.

```

data class FullName(
    val firstName: String,
    val secondName: String,
    val lastName: String
)

val elon = FullName("Elon", "Reeve", "Musk")
val (name, surname) = elon
print("It is $name $surname!") // It is Elon Reeve!

```

We need to be careful with destructuring. It is useful to use the same names as data class primary constructor properties. In the case of an incorrect order, an IntelliJ/Android Studio warning will be shown. It might even be useful to upgrade this warning to an error.

```

data class FullName(
    val firstName: String,
    val secondName: String,
    val lastName: String
)

val elon = FullName("Elon", "Reeve", "Musk")
val (firstName, lastName) = elon
print("It is $name $lastName") // It is Elon Reeve!

```

Variable name 'lastName' matches the name of a different component more... (⌘F1)

Destructuring a single value in lambda is very confusing, especially since parentheses around arguments in lambda expressions are either optional or required in some languages.

```

data class User(
    val name: String,
    val surname: String,
)

fun main() {
    val users = listOf(
        User("Nicola", "Corti")
    )
    users.forEach { u -> println(u) }
    // User(name=Nicola, surname=Corti)
}

```

```
users.forEach { (u) -> println(u) }
// Nicola
}
```

Data class limitations

The idea behind data classes is that they represent a bundle of data; their constructors allow us to specify all this data, and we can access it through destructuring or by copying them to another instance with the `copy` method. This is why only primary constructor properties are considered by the methods defined in data classes.

```
data class Dog(
    val name: String,
) {
    // Bad practice, avoid mutable properties in data classes
    var trained = false
}

fun main() {
    val d1 = Dog("Cookie")
    d1.trained = true
    println(d1) // Dog(name=Cookie)
    // so nothing about trained property

    val d2 = d1.copy()
    println(d1.trained) // true
    println(d2.trained) // false
    // so trained value not copied
}
```

Data classes are supposed to keep all the essential properties in their primary constructor. Inside the body, we should only keep redundant immutable properties, which means properties whose value is distinctly calculated from primary constructor properties, like `fullName`, which is calculated from `name` and `surname`. Such values are also ignored by data class

methods, but their value will always be correct because it will be calculated when a new object is created.

```
data class FullName(  
    val name: String,  
    val surname: String,  
) {  
    val fullName = "$name $surname"  
}  
  
fun main() {  
    val d1 = FullName("Cookie", "Moskała")  
    println(d1.fullName) // Cookie Moskała  
    println(d1) // FullName(name=Cookie, surname=Moskała)  
  
    val d2 = d1.copy()  
    println(d2.fullName) // Cookie Moskała  
    println(d2) // FullName(name=Cookie, surname=Moskała)  
}
```

You should also remember that data classes must be **final** and so cannot be used as a super-type for inheritance.

Prefer data classes instead of tuples

Data classes offer more than what is generally provided by tuples. Historically, they replaced tuples in Kotlin since they are considered better practice⁵¹. The only tuples that are left are `Pair` and `Triple`, but these are data classes under the hood:

⁵¹Kotlin had support for tuples when it was still in the beta version. We were able to define a tuple by brackets and a set of types, like `(Int, String, String, Long)`. What we achieved behaved the same as data classes in the end, but it was far less readable. Can you guess what type this set of types represents? It can be anything. Using tuples is tempting, but using data classes is nearly always better. This is why tuples were removed, and only `Pair` and `Triple` are left.

```
data class Pair<out A, out B>(
    val first: A,
    val second: B
) : Serializable {

    override fun toString(): String =
        "($first, $second)"
}

data class Triple<out A, out B, out C>(
    val first: A,
    val second: B,
    val third: C
) : Serializable {

    override fun toString(): String =
        "($first, $second, $third)"
}
```

The easiest way to create a `Pair` is by using the `to` function. This is a generic infix extension function, defined as follows (we will discuss both generic and extension functions in later chapters).

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

Thanks to the infix modifier, a method can be used by placing its name between arguments, as the infix name suggests. The result `Pair` is typed, so the result type from the "ABC" to 123 expression is `Pair<String, Int>`.

```
fun main() {
    val p1: Pair<String, Int> = "ABC" to 123
    println(p1) // (ABC, 123)
    val p2 = 'A' to 3.14
    // the type of p2 is Pair<Char, Double>
    println(p2) // (A, 123)
    val p3 = true to false
    // the type of p3 is Pair<Boolean, Boolean>
    println(p3) // (true, false)
}
```

These tuples remain because they are very useful for local purposes, like:

- When we immediately name values:

```
val (description, color) = when {
    degrees < 5 -> "cold" to Color.BLUE
    degrees < 23 -> "mild" to Color.YELLOW
    else -> "hot" to Color.RED
}
```

- To represent an aggregate that is not known in advance, as is commonly the case in standard library functions:

```
val (odd, even) = numbers.partition { it % 2 == 1 }
val map = mapOf(1 to "San Francisco", 2 to "Amsterdam")
```

In other cases, we prefer data classes. Take a look at an example: let's say that we need a function that parses a full name into a name and a surname. One might represent this name and surname as a `Pair<String, String>`:

```
fun String.parseName(): Pair<String, String>? {
    val indexOfLastSpace = this.trim().lastIndexOf(' ')
    if (indexOfLastSpace < 0) return null
    val firstName = this.take(indexOfLastSpace)
    val lastName = this.drop(indexOfLastSpace)
    return Pair(firstName, lastName)
}

// Usage
fun main() {
    val fullName = "Marcin Moskała"
    val (firstName, lastName) = fullName.parseName() ?: return
}
```

The problem is that when someone reads this code, it is not clear that `Pair<String, String>` represents a full name. What is more, it is not clear what the order of the values is, therefore someone might think that the surname goes first:

```
val fullName = "Marcin Moskała"
val (lastName, firstName) = fullName.parseName() ?: return
print("His name is $firstName") // His name is Moskała
```

To make usage safer and the function easier to read, we should use a data class instead:

```
data class FullName(
    val firstName: String,
    val lastName: String
)

fun String.parseName(): FullName? {
    val indexOfLastSpace = this.trim().lastIndexOf(' ')
    if (indexOfLastSpace < 0) return null
    val firstName = this.take(indexOfLastSpace)
    val lastName = this.drop(indexOfLastSpace)
    return FullName(firstName, lastName)
}
```

```
// Usage
fun main() {
    val fullName = "Marcin Moskała"
    val (firstName, lastName) = fullName.parseName() ?: return
    print("His name is $firstName $lastName")
    // His name is Marcin Moskała
}
```

This costs nearly nothing and improves the function significantly:

- The return type of this function is more clear.
- The return type is shorter and easier to pass forward.
- If a user destructures variables with correct names but in incorrect positions, a warning will be displayed in IntelliJ.

If you don't want this class in a wider scope, you can restrict its visibility. It can even be private if you only need to use it for some local processing in a single file or class. It is worth using data classes instead of tuples. Classes are cheap in Kotlin, so don't be afraid to use them in your projects.

Summary

In this chapter, we've learned about `Any`, which is a superclass of all classes. We've also learned about methods defined by `Any`: `equals`, `hashCode`, and `toString`. We've also learned that there are two primary types of objects. Regular objects are considered unique and do not expose their details. Data class objects, which we made using the `data` modifier, represent bundles of data (we keep them in primary constructor properties). They are equal when they hold the same data. When transformed to a string, they print all their data. They additionally support destructuring and making a copy with the `copy` method. Two generic data classes in Kotlin stdlib are `Pair` and `Triple`, but (apart from certain cases) we prefer to use custom data classes

instead of these. Also, for the sake of safety, when we destructure a data class, we prefer to match the variable names with the parameter names.

Now, let's move on to a topic dedicated to special Kotlin syntax that lets us create objects without defining a class.

Objects

What is an object? This is the question I often start this section with in my workshops, and I generally get an instant response, “An instance of a class”. That is right, but how do we create objects? One way is easy: using constructors.

```
class A

// Using a constructor to create an object
val a = A()
```

However, this is not the only way. In Kotlin, we can also create objects using **object expression** and **object declaration**. Let’s discuss these two options.

Object expressions

To create an empty object using an expression, we use the `object` keyword and braces. This syntax for creating objects is known as **object expression**.

```
val instance = object {}
```

An empty object extends no classes (except for `Any`, which is extended by all objects in Kotlin), implements no interfaces, and has nothing inside its body. Nevertheless, it is useful. Its power lies in its uniqueness: such an object equals nothing else but itself. Therefore, it is perfectly suited to be used as some kind of token or synchronization lock.

```
class Box {
    var value: Any? = NOT_SET

    fun initialized() = value != NOT_SET

    companion object {
        private val NOT_SET = object {}
    }
}

private val LOCK = object {}

fun synchronizedOperation() = synchronized(LOCK) {
    // ...
}
```

An empty object can also be created with the constructor of `Any`, so `Any()` is an alternative to `object {}`.

```
private val NOT_SET = Any()
```

However, objects created with an object expression do not need to be empty. They can have bodies, extend classes, implement interfaces, etc. The syntax is the same as for classes, but object declarations use the `object` keyword instead of `class` and should not define the name or constructor.

```
data class User(val name: String)

interface UserProducer {
    fun produce(): User
}

fun printUser(producer: UserProducer) {
    println(producer.produce())
}

fun main() {
    val user = User("Jake")
```

```
val producer = object : UserProducer {
    override fun produce(): User = user
}
printUser(producer) // User(name=Jake)
}
```

In a local scope, object expressions define an anonymous type that won't work outside the class where it is defined. This means the non-inherited members of object expressions are accessible only when an anonymous object is declared in a local or class-private scope; otherwise, the object is just an opaque Any type, or the type of the class or interface it inherits from. This makes non-inherited members of object expressions hard to use in real-life projects.

```
class Robot {
    // Possible, but rarely useful
    // prefer regular member properties instead
    private val point = object {
        var x = 0
        var y = 0
    }

    fun moveUp() {
        point.y += 10
    }

    fun show() {
        println("${point.x}, ${point.y}")
    }
}

fun main() {
    val robot = Robot()
    robot.show() // (0, 0)
    robot.moveUp()
    robot.show() // (0, 10)

    val point = object {
```

```
    var x = 0
    var y = 0
}
println(point.x) // 0
point.y = 10
println(point.y) // 10
}
```

In practice, object expressions are used as an alternative to Java anonymous classes, i.e., when we need to create a watcher or a listener with multiple handler methods.

```
taskNameView.addTextChangedListener(object : TextWatcher {
    override fun afterTextChanged(
        editable: Editable?
    ) {
        //...
    }

    override fun beforeTextChanged(
        text: CharSequence?,
        start: Int,
        count: Int,
        after: Int
    ) {
        //...
    }

    override fun onTextChanged(
        text: CharSequence?,
        start: Int,
        before: Int,
        count: Int
    ) {
        //...
    }
})
```

Note that “object expression” is a better name than “anony-

mous class” since this is an expression that produces an object.

Object declaration

If we take an object expression and give it a name, we get an **object declaration**. This structure also creates a single object, but this object is not anonymous: it has a name that can be used to reference it.

```
object Point {  
    var x = 0  
    var y = 0  
}  
  
fun main() {  
    println(Point.x) // 0  
    Point.y = 10  
    println(Point.y) // 10  
  
    val p = Point  
    p.x = 20  
    println(Point.x) // 20  
    println(Point.y) // 10  
}
```

Object declaration is an implementation of a singleton pattern⁵², so this declaration creates a class with a single instance. Whenever we want to use this class, we need to operate on this single instance. Object declarations support all the features that classes support; for example, they can extend classes or implement interfaces.

⁵²A programming pattern where a class is implemented such that it can have only one instance.

```
data class User(val name: String)

interface UserProducer {
    fun produce(): User
}

object FakeUserProducer : UserProducer {
    override fun produce(): User = User("fake")
}

fun setUserProducer(producer: UserProducer) {
    println(producer.produce())
}

fun main() {
    setUserProducer(FakeUserProducer) // User(name=fake)
}
```

Companion objects

When I reflect on the times when I worked as a Java developer, I remember discussions about what features should be introduced into that language. A common idea I often heard was introducing inheritance for static elements. In the end, inheritance is very important in Java, so why can't we use it for static elements? Kotlin has addressed this problem with companion objects; however, to make that possible, it first needed to eliminate actual static elements, i.e., elements that are called on classes, not on objects.

```
// Java
class User {
    // Static element definition
    public static User empty() {
        return new User();
    }
}

// Static element usage
User user = User.empty()
```

Yes, we don't have static elements in Kotlin, but we don't need them because we use object declarations instead. If we define an object declaration in a class, it is static by default (just like classes defined inside classes), so we can directly call its elements.

```
// Kotlin
class User {
    object Producer {
        fun empty() = User()
    }
}

// Usage
val user: User = User.Producer.empty()
```

This is not as convenient as static elements, but we can improve it. If we use the `companion` keyword before an object declaration defined inside a class, then we can call these object methods implicitly “on the class”.

```
class User {  
    companion object Producer {  
        fun empty() = User()  
    }  
}  
  
// Usage  
val user: User = User.empty()  
// or  
val user: User = User.Producer.empty()
```

Objects with the `companion` modifier, also known as companion objects, do not need an explicit name. Their default name is `Companion`.

```
class User {  
    companion object {  
        fun empty() = User()  
    }  
}  
  
// Usage  
val user: User = User.empty()  
// or  
val user: User = User.Companion.empty()
```

This is how we achieved a syntax that is nearly as convenient as static elements. The only inconvenience is that we must locate all the “static” elements inside a single object (there can be only one companion object in a class). This is a limitation, but we have something in return: companion objects are objects, so they can extend classes or implement interfaces.

Let me show you an example. Let’s say that you represent money in different currencies using different classes like `USD`, `EUR`, or `PLN`. For convenience, each of these defines `fromBuilder` functions, which simplify object creation.

```
import java.math.BigDecimal
import java.math.MathContext
import java.math.RoundingMode.HALF_EVEN

abstract class Money(
    val amount: BigDecimal,
    val currency: String
)

class USD(amount: BigDecimal) : Money(amount, "USD") {
    companion object {
        private val MATH = MathContext(2, HALF_EVEN)
        fun from(amount: Int): USD =
            USD(amount.toBigDecimal(MATH))
        fun from(amount: Double): USD =
            USD(amount.toBigDecimal(MATH))

        @Throws(NumberFormatException::class)
        fun from(amount: String): USD =
            USD(amount.toBigDecimal(MATH))
    }
}

class EUR(amount: BigDecimal) : Money(amount, "EUR") {
    companion object {
        private val MATH = MathContext(2, HALF_EVEN)
        fun from(amount: Int): EUR =
            EUR(amount.toBigDecimal(MATH))
        fun from(amount: Double): EUR =
            EUR(amount.toBigDecimal(MATH))

        @Throws(NumberFormatException::class)
        fun from(amount: String): EUR =
            EUR(amount.toBigDecimal(MATH))
    }
}

class PLN(amount: BigDecimal) : Money(amount, "PLN") {
    companion object {
```

```
private val MATH = MathContext(2, HALF_EVEN)
fun from(amount: Int): PLN =
    PLN(amount.toBigDecimal(MATH))
fun from(amount: Double): PLN =
    PLN(amount.toBigDecimal(MATH))

@Throws(NumberFormatException::class)
fun from(amount: String): PLN =
    PLN(amount.toBigDecimal(MATH))
}

fun main() {
    val eur: EUR = EUR.from("12.00")
    val pln: PLN = PLN.from(20)
    val usd: USD = USD.from(32.5)
}
```

The repetitive functions for creating objects from different types can be extracted into an abstract `MoneyMaker` class, which can be extended by companion objects of different currencies. This class can offer a range of methods to create a currency. This way, we use companion object inheritance to extract a pattern that is common to all companion objects of classes that represent money.

```
import java.math.BigDecimal
import java.math.MathContext
import java.math.RoundingMode.HALF_EVEN

abstract class Money(
    val amount: BigDecimal,
    val currency: String
)

abstract class MoneyMaker<Currency : Money> {
    private val MATH = MathContext(2, HALF_EVEN)
    abstract fun from(amount: BigDecimal): Currency
    fun from(amount: Int): Currency =
```

```
    from(amount.toBigDecimal(MATH))
fun from(amount: Double): Currency =
    from(amount.toBigDecimal(MATH))

@Throws(NumberFormatException::class)
fun from(amount: String): Currency =
    from(amount.toBigDecimal(MATH))
}

class USD(amount: BigDecimal) : Money(amount, "USD") {
    companion object : MoneyMaker<USD>() {
        override fun from(amount: BigDecimal): USD =
            USD(amount)
    }
}

class EUR(amount: BigDecimal) : Money(amount, "EUR") {
    companion object : MoneyMaker<EUR>() {
        override fun from(amount: BigDecimal): EUR =
            EUR(amount)
    }
}

class PLN(amount: BigDecimal) : Money(amount, "PLN") {
    companion object : MoneyMaker<PLN>() {
        override fun from(amount: BigDecimal): PLN =
            PLN(amount)
    }
}

fun main() {
    val eur: EUR = EUR.from("12.00")
    val pln: PLN = PLN.from(20)
    val usd: USD = USD.from(32.5)
}
```

Our community is still learning how to use these capabilities, but you can already find plenty of examples in projects and

libraries. Here are a few interesting examples⁵³:

```
// Using companion object inheritance for logging
// from the Kotlin Logging framework
class FooWithLogging {
    fun bar(item: Item) {
        logger.info { "Item $item" }
        // Logger comes from the companion object
    }

    companion object : KLogging()
    // companion inherits logger property
}

// Android-specific example of using an abstract factory
// for companion object
class MainActivity : Activity() {
    //...

    // Using companion object as a factory
    companion object : ActivityFactory() {
        override fun getIntent(context: Context): Intent =
            Intent(context, MainActivity::class.java)
    }
}

abstract class ActivityFactory {
    abstract fun getIntent(context: Context): Intent

    fun start(context: Context) {
        val intent = getIntent(context)
        context.startActivity(intent)
    }
}
```

⁵³Do not treat them as best practices but rather as examples of what you might do with the fact that companion objects can inherit from classes and implement interfaces.

```
fun startForResult(activity: Activity, requestCode: Int) {
    val intent = getIntent(activity)
    activity.startActivityForResult(intent, requestCode)
}

// Usage of all the members of the companion ActivityFactory
val intent = MainActivity.getIntent(context)
MainActivity.start(context)
MainActivity.startForResult(activity, requestCode)

// In contexts on Kotlin Coroutines, companion objects are
// used as keys to identify contexts
data class CoroutineName(
    val name: String
) : AbstractCoroutineContextElement(CoroutineName) {

    // Companion object is a key
    companion object Key : CoroutineContext.Key<CoroutineName>

    override fun toString(): String = "CoroutineName($name)"
}

// Finding a context by key
val name1 = context[CoroutineName] // Yes, this is a companion

// You can also refer to companion objects by its name
val name2 = context[CoroutineName.Key]
```

Data object declarations

Since Kotlin 1.8, you can use the `data` modifier for object declarations. It generates the `toString` method for the object; this method includes the object name as a string.

```
data object ABC

fun main() {
    println(ABC) // ABC
}
```

Constant values

It's common practice to generally extract constant values as properties of companion objects and name them using UPPER_SNAKE_CASE⁵⁴. This way, we name those values and simplify their changes in the future. We name constant values in a characteristic way to make it clear that they represent a constant⁵⁵.

```
class Product(
    val code: String,
    val price: Double,
) {
    init {
        require(price > MIN_AMOUNT)
    }

    companion object {
        val MIN_AMOUNT = 5.00
    }
}
```

When companion object properties or top-level properties represent a constant value (known at compile time) that is

⁵⁴UPPER_SNAKE_CASE is a naming convention where each character is capitalized, and we separate words with an underscore, like in the UPPER_SNAKE_CASE name. Using it for constants is suggested in the Kotlin documentation in the section *Kotlin Coding Convention*.

⁵⁵This practice is better described in Effective Kotlin, Item 27: *Use abstraction to protect code against changes*.

either a primitive or a `String`⁵⁶, we can add the `const` modifier. This is an optimization. All usages of such variables will be replaced with their values at compile time.

```
class Product(  
    val code: String,  
    val price: Double,  
) {  
    init {  
        require(price > MIN_AMOUNT)  
    }  
  
    companion object {  
        const val MIN_AMOUNT = 5.00  
    }  
}
```

Such properties can also be used in annotations:

```
private const val OUTDATED_API: String =  
    "This is a part of an outdated API."  
  
@Deprecated(OUTDATED_API)  
fun foo() {  
    ...  
}  
  
@Deprecated(OUTDATED_API)  
fun boo() {  
    ...  
}
```

Summary

In this chapter, we've learned that objects can be created not only from classes but also using object expressions and

⁵⁶So, the accepted types are `Int`, `Long`, `Double`, `Float`, `Short`, `Byte`, `Boolean`, `Char`, and `String`.

object declarations. Both these kinds of objects have practical usages. Object expression is used as an alternative to Java anonymous objects, but it offers more. Object declaration is Kotlin's implementation of the singleton pattern. A special form of object declaration, known as a companion object, is used as an alternative to static elements but with additional support for inheritance. We also have the `const` modifier, which offers better support for constant elements defined at the top level or in object declarations.

In the previous chapter, we discussed data classes, but there are other modifiers we use for classes in Kotlin. In the next chapter, we will learn about another important type of class: exceptions.

Exceptions

An exception is a generally unwanted event that interrupts the regular flow of your program. It might occur when you perform an illegal operation. Exceptions contain information that helps developers find out what led to this problem.

Let's take a look at an example. When you divide an integer by 0, an exception of type `ArithmeticException` will be thrown. Each exception might have a message included that should explain what went wrong. In this case, the message will be “/ by zero”. Each exception also includes its stack trace, which is a list of the method calls that the application was in the middle of when the exception was thrown. In this example, it includes information that this exception was thrown from the `calculate` function, which was called from the `printCalculated` function, which was called from the `main` function. Exception interrupts program execution, so statements after it won't be executed. In the example below, notice that “After” is never printed.

```
private fun calculate(): Int {
    return 1 / 0
}

private fun printCalculated() {
    println(calculate())
}

fun main() {
    println("Before")
    printCalculated()
    println("After")
}
// Before
// Exception java.lang.ArithmeticException: / by zero
//     at PlaygroundKt.calculate(Playground.kt:2)
//     at PlaygroundKt.printCalculated(Playground.kt:6)
```

```
//      at PlaygroundKt.main(Playground.kt:11)
//      at PlaygroundKt.main(Playground.kt)
```

As another example, we can parse a string to an integer using the `toInt` method, but this only works when the string is a number. When it isn't, we will see `NumberFormatException` with a message explaining which string was used.

```
fun main() {
    val i1 = "10".toInt()
    println(i1)
    val i2 = "ABC".toInt()
    println(i2)
}
// 10
// Exception in thread "main" java.lang.NumberFormatException:
// For input string: "ABC"
//  at java.base/java.lang.NumberFormatException.
//  forInputString(NumberFormatException.java:67)
//  at java.base/java.lang.Integer.parseInt(Integer.java:660)
//  at java.base/java.lang.Integer.parseInt(Integer.java:778)
//  at PlaygroundKt.main(Playground.kt:4)
//  at PlaygroundKt.main(Playground.kt)
```

Throwing exceptions

We can throw exceptions ourselves using the `throw` keyword and a value that can be used as an exception, like the aforementioned `ArithmaticException` or `NumberFormatException`.

```
private fun functionThrowing() {
    throw ArithmeticException("Some message")
}

fun main() {
    println("Before")
    functionThrowing()
    println("After")
}
// Before
// Exception in thread "main" java.lang.ArithmetiException:
// Some message
// at PlaygroundKt.functionThrowing(Playground.kt:2)
// at PlaygroundKt.main(Playground.kt:7)
// at PlaygroundKt.main(Playground.kt)
```

Exceptions communicate conditions that a function is not prepared to handle or is not responsible for. This isn't necessarily an indication of an error; it's more like a notification event that can be dealt with in another place that is set up to catch it.

Defining exceptions

We can also define our own exceptions. These are regular classes or object declarations that extend the `Throwable` class. Every such class can be thrown using `throw`.

```
class MyException : Throwable("Some message")
object MyExceptionObject : Throwable("Some message")

private fun functionThrowing() {
    throw MyException()
    // or throw MyExceptionObject
}

fun main() {
    println("Before")
```

```
    functionThrowing()
    println("After")
}
// Before
// Exception in thread "main" MyException: Some message
// at PlaygroundKt.functionThrowing(Playground.kt:4)
// at PlaygroundKt.main(Playground.kt:9)
// at PlaygroundKt.main(Playground.kt)
```

Catching exceptions

Just like exceptions can be thrown, they can be caught using a try-catch structure that contains a try-block and a catch-block. An exception thrown in a function immediately ends this function's execution, and the process repeats in the function that called the function in which the exception was thrown. This changes when an exception is thrown inside a try-block, because then its catch-blocks are checked. Each catch-block can specify what type of exceptions it catches. The first catch-block that accepts the exception that was thrown will catch it and then execute its body. If an exception is caught, the execution of the program continues after the try block.

```
class MyException : Throwable("Some message")

fun someFunction() {
    throw MyException()
    println("Will not be printed")
}

fun main() {
    try {
        someFunction()
        println("Will not be printed")
    } catch (e: MyException) {
        println("Caught $e")
        // Caught MyException: Some message
    }
}
```

```
    }  
}
```

Let's see try-catch with more catch-blocks in action. Remember that the first block that accepts an exception is always chosen. A catch-block accepts an exception if this exception is a subtype of the type specified in the catch-block. Note that all exceptions must extend `Throwable`, so catching this type means catching all possible exceptions.

```
import java.lang.NumberFormatException  
  
class MyException : Throwable("Some message")  
  
fun testTryCatch(exception: Throwable) {  
    try {  
        throw exception  
    } catch (e: ArithmeticException) {  
        println("Got ArithmeticException")  
    } catch (e: MyException) {  
        println("Got MyException")  
    } catch (e: Throwable) {  
        println("Got some exception")  
    }  
}  
  
fun main() {  
    testTryCatch(ArithmeticException())  
    // Got ArithmeticException  
    testTryCatch(MyException())  
    // Got MyException  
    testTryCatch(NumberFormatException())  
    // Got some exception  
}
```

Try-catch block used as an expression

The try-catch structure can be used as an expression. It returns the result of a try-block if no exception occurred. If an

exception occurs and is caught, then the try-catch expression returns the result of the catch-block.

```
fun main() {
    val a = try {
        1
    } catch (e: Error) {
        2
    }
    println(a) // 1

    val b = try {
        throw Error()
        1
    } catch (e: Error) {
        2
    }
    println(b) // 2
}
```

A try-catch expression can be used to provide an alternative value for a situation in which a problem occurs:

```
import java.io.File
import java.io.FileNotFoundException

fun main() {
    val content = try {
        File("AAA").readText()
    } catch (e: FileNotFoundException) {
        ""
    }
    println(content) // (empty string)
}
```

A practical example might be reading a string containing an object in JSON format. We use the Gson library, whose `fromJson` method throws `JsonSyntaxException` when a string

does not contain a proper JSON object. Instead, we would prefer a function that returns `null` in such cases; we can implement this using try-catch as an expression.

```
fun <T : Any> fromJsonOrNull(
    json: String,
    clazz: KClass<T>
): T? = try {
    gson.fromJson(json, clazz.java)
} catch (e: JsonSyntaxException) {
    null
}
```

Finally block

Inside the try-structure, we can also use a finally-block, which is used to specify what should always be invoked, even if an exception occurs. This block does not catch any exceptions; it is used to guarantee that some operations will be executed, no matter the exceptions.

Take a look at the code below. An exception is thrown inside `someFunction`. This exception ends this function's execution and skips the rest of the try-block. Since we do not have a catch-block, this exception will not be caught, thus ending the execution of the `main` function. However, there is also the finally-block, whose body is invoked even if an exception occurs.

```
fun someFunction() {
    throw Throwable("Some error")
}

fun main() {
    try {
        someFunction()
    } finally {
        println("Finally block was called")
}
```

```
        }
        println("Will not be printed")
    }
// Finally block was called
```

The finally-block is also invoked when the try-block finishes without an exception.

```
fun someFunction() {
    println("Function called")
}

fun main() {
    try {
        someFunction()
        println("After call")
    } finally {
        println("Finally block was called")
    }
    println("After try-finally")
}
// Function called
// After call
// Finally block was called
// After try-finally
```

We use the finally-block to do operations that should always be done, no matter if an exception occurs or not. It typically involves closing connections or cleaning up resources.

Important exceptions

A few kinds of exceptions are defined in Kotlin that we use in certain situations. The most important ones are:

- `IllegalArgumentException` - we use this when an argument has an incorrect value. For example, when you expect your argument value to be bigger than 0 but it is not.

- `IllegalStateException` - we use this when the state of our system is incorrect. This means the values of properties are not accepted by a function call.

```
fun findClusters(number: Int) {  
    if (number < 1) {  
        throw IllegalArgumentException("...")  
    }  
    // ...  
}  
  
var userName = ""  
  
fun printUserName() {  
    if (userName == "") {  
        throw IllegalStateException("Name must not be empty")  
    }  
    // ...  
}
```

In Kotlin, we use the `require` and `check` functions to throw `IllegalArgumentException` and `IllegalStateException` when their conditions are not satisfied⁵⁷.

```
fun pop(num: Int): List<T> {  
    require(num <= size)  
    // throws IllegalArgumentException if num > size  
    check(isOpen)  
    // throws IllegalStateException if is not open  
    val ret = collection.take(num)  
    collection = collection.drop(num)  
    return ret  
}
```

⁵⁷This topic is better described in the Effective Kotlin book, Item 5: Specify your expectations for arguments and states.

There is also an `error` function from Kotlin stdlib that throws `IllegalArgumentException` with a message specified as an argument. It is often used as a body for a branch in a `when`-condition, on the right side of the Elvis operator, or in an if-else expression.

```
fun makeOperation(  
    operation: String,  
    left: Int,  
    right: Int? = null  
) : Int = when (operation) {  
    "add" ->  
        left + (right ?: error("Two numbers required"))  
    "subtract" ->  
        left - (right ?: error("Two numbers required"))  
    "opposite" -> -left  
    else -> error("Unknown operation")  
}  
  
fun main() {  
    println(makeOperation("add", 1, 2)) // 3  
    println(makeOperation("subtract", 1, 2)) // -1  
    println(makeOperation("opposite", 10)) // -10  
  
    makeOperation("add", 1) // ERROR!  
    // IllegalStateException: Two numbers required  
    makeOperation("subtract", 1) // ERROR!  
    // IllegalStateException: Two numbers required  
    makeOperation("other", 1, 2) // ERROR!  
    // IllegalStateException: Unknown operation  
}
```

Hierarchy of exceptions

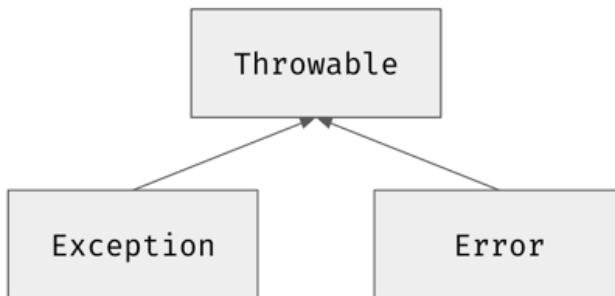
The most important subtypes of `Throwable` are `Error` and `Exception`. These represent two types of exceptions:

- `Error` type represents exceptions that are impossible to recover from and consequently should not be caught, at

least not without throwing them again in catch-block. Exceptions that are impossible to recover from includes `OutOfMemoryError`, which is thrown when there is insufficient space in the JVM heap.

- `Exception` type represents exceptions we can recover from using a try-catch block. This group includes `IllegalArgumentException`, `IllegalStateException`, `ArithmetricException`, and `NumberFormatException`.

In most cases, when we define custom exceptions, we should use the `Exception` superclass; when we catch exceptions, we should only throw subtypes of `Exception`.



In Kotlin, we are not forced to catch any kinds of exceptions; so, unlike in some other languages, there are no checked exceptions.

Summary

In this chapter we've learned about exceptions, which are an important part of Kotlin programming. We've learned how to throw, catch, and define exceptions. We've also learned about the finally-block and the exceptions hierarchy.

Continuing with special kinds of classes, let's talk about enum classes, which are used to represent a set of object instance values.

Enum classes

In this chapter, we're going to introduce the concept of enum classes. Let's start with an example. Suppose that you're implementing a payment method that has to support three possible options: cash payment, card payment, and bank transfer. The most basic way to represent a fixed set of values in Kotlin is an enum class. Inside its body, we list all the values, divided by a comma. We name values using UPPER_SNAKE_CASE notation (e.g., BANK_TRANSFER). Enum class elements can be referenced by the enum name, followed by a dot, and then the value name (e.g., PaymentOption.CASH). All values are typed as the enum class type.

```
enum class PaymentOption {
    CASH,
    CARD,
    TRANSFER,
}

fun printOption(option: PaymentOption) {
    println(option)
}

fun main() {
    val option: PaymentOption = PaymentOption.CARD
    println(option) // CARD
    printOption(option) // CARD
}
```

Each enum class has the following companion object functions:

- `values`, which returns an array of all the values of this enum class;
- `valueOf`, which parses a string into a value matching its name (this is case-sensitive) or throws an exception.

```
enum class PaymentOption {
    CASH,
    CARD,
    TRANSFER,
}

fun main() {
    val option: PaymentOption =
        PaymentOption.valueOf("TRANSFER")
    println(option)

    println("All options: ")
    val paymentOptions: Array<PaymentOption> =
        PaymentOption.values()
    for (paymentOption in paymentOptions) {
        println(paymentOption)
    }
}
// TRANSFER
// All options:
// CASH
// CARD
// TRANSFER
```

Instead of these methods, we can also use the top-level `enumValues` and `enumValueOf` functions.

```
enum class PaymentOption {
    CASH,
    CARD,
    TRANSFER,
}

fun main() {
    val option = enumValueOf<PaymentOption>("TRANSFER")
    println(option)

    println("All options: ")
    val paymentOptions = enumValues<PaymentOption>()
```

```
    for (paymentOption in paymentOptions) {
        println(paymentOption)
    }
}

// TRANSFER
// All options:
// CASH
// CARD
// TRANSFER
```

As you can see, enum elements keep their values in order. This order is important. Each enum value has two properties:

- `name` - the name of this value,
- `ordinal` - the position of this value (starting from 0).

```
enum class PaymentOption {
    CASH,
    CARD,
    TRANSFER,
}

fun main() {
    val option = PaymentOption.TRANSFER
    println(option.name) // TRANSFER
    println(option.ordinal) // 2
}
```

Each enum class is a subclass of the abstract class `Enum`. This superclass guarantees the `name` and `ordinal` properties. `Enum` classes have properties that implement `toString`, `equals`, and `hashCode`, but, unlike data classes, they also have `compareTo` (their natural order is the order of the elements in the body).

Enum values can be used inside when-conditions. Moreover, there is no need to use the else-branch when all possible enum values are covered.

```
fun transactionFee(paymentOption: PaymentOption): Double =  
    when (paymentOption) {  
        PaymentOption.CASH -> 0.0  
        PaymentOption.CARD, PaymentOption.TRANSFER -> 0.05  
    }
```

Enum classes are very convenient because they can be easily parsed or stringified. They are a popular way to represent a finite set of possible values.

Data in enum values

In Kotlin, each enum value can hold a state. It is possible to define a primary constructor for an enum class, and then each value needs to specify its data next to its name. **It is a best practice that enum values should always be immutable, so their state should never change.**

```
import java.math.BigDecimal  
  
enum class PaymentOption(val commission: BigDecimal) {  
    CASH(BigDecimal.ONE),  
    CARD(BigDecimal.TEN),  
    TRANSFER(BigDecimal.ZERO)  
}  
  
fun main() {  
    println(PaymentOption.CARD.commission) // 10  
    println(PaymentOption.TRANSFER.commission) // 0  
  
    val paymentOption: PaymentOption =  
        PaymentOption.values().random()  
    println(paymentOption.commission) // 0, 1 or 10  
}
```

Enum classes with custom methods

Kotlin enums can have abstract methods whose implementations are item-specific. When we define them, the enum class itself needs to define an abstract method, and each item must override it:

```
enum class PaymentOption {
    CASH {
        override fun startPayment(
            transaction: Transaction
        ) {
            showCashPaymentInfo(transaction)
        }
    },
    CARD {
        override fun startPayment(
            transaction: Transaction
        ) {
            moveToCardPaymentPage(transaction)
        }
    },
    TRANSFER {
        override fun startPayment(
            transaction: Transaction
        ) {
            showMoneyTransferInfo()
            setupPaymentWatcher(transaction)
        }
    };
}

abstract fun startPayment(transaction: Transaction)
}
```

This option is not popular as we generally prefer using func-

tional primary constructor parameters⁵⁸ or extension functions⁵⁹.

Summary

Enum classes are a convenient way to represent a concrete set of possible values. Each value has the properties `name` and `ordinal` (position). We can get an array of all values using the `values` companion object function or the `enumValues` top-level function. We can also parse an enum value from `String` using the `valueOf` companion object function or the `enumValueOf` top-level function.

In the next chapter, we will talk about sealed classes, which are often treated as similar to enums but represent completely different and even more powerful abstractions. Sealed classes can form a closed hierarchy of classes, whereas enums represent only a set of constant values.

⁵⁸Functional variables are described in the book *Functional Kotlin*. An example of using an enum class with functional primary constructor parameters is presented in *Effective Kotlin*, Item 41: *Use enum to represent a list of values*.

⁵⁹Extension functions are described later in this book.

Sealed classes and interfaces

Classes and interfaces in Kotlin are not only used to represent a set of operations or data; we can also use classes and inheritance to represent hierarchies through polymorphism. For instance, let's say that you send a network request; as a result, you either successfully receive the requested data, or the request fails with some information about what went wrong. These two outcomes can be represented using two classes that implement an interface:

```
interface Result
class Success(val data: String) : Result
class Failure(val exception: Throwable) : Result
```

Alternatively, you could use an abstract class:

```
abstract class Result
class Success(val data: String) : Result()
class Failure(val exception: Throwable) : Result()
```

With either of these, we know that when a function returns `Result`, it can be `Success` or `Failure`.

```
val result: Result = getSomeData()
when (result) {
    is Success -> handleSuccess(result.data)
    is Failure -> handleFailure(result.exception)
}
```

The problem is that when a regular interface or abstract class is used, there is no guarantee that its defined subclasses are all possible subtypes of this interface or abstract class. Someone might define another class and make it implement `Result`. Someone might also implement an object expression that implements `Result`.

```
class FakeSuccess : Result

val res: Result = object : Result {}
```

A hierarchy whose subclasses are not known in advance is known as a non-restricted hierarchy. For `Result`, we prefer to define a restricted hierarchy, which we do by using a `sealed` modifier before a class or an interface⁶⁰.

```
sealed interface Result
class Success(val data: String) : Result
class Failure(val exception: Throwable) : Result

// or

sealed class Result
class Success(val data: String) : Result()
class Failure(val exception: Throwable) : Result()
```

When we use the `sealed` modifier before a class, it makes this class abstract already, so we don't use the `abstract` modifier.

There are a few requirements that all sealed class or interface children must meet:

- they need to be defined in the same package and module where the sealed class or interface is,
- they can't be local or defined using object expression.

This means that when you use the `sealed` modifier, you control which subclasses a class or interface has. The clients

⁶⁰Restricted hierarchies are used to represent values that could take on several different but fixed types. In other languages, restricted hierarchies might be represented by sum types, coproducts, or tagged unions.

of your library or module cannot add their own direct subclasses⁶¹. No one can quietly add a local class or object expression that extends a sealed class or interface. Kotlin has made this impossible. The hierarchy of subclasses is restricted.

Sealed interfaces were introduced in more recent versions of Kotlin to allow classes to implement multiple sealed hierarchies. The relation between a sealed class and a sealed interface is similar to the relation between an abstract class and an interface. The power of classes is that they can keep a state (non-abstract properties) and control their members' openness (can have final methods and properties). The power of interfaces is that a class can inherit from only one class but it can implement multiple interfaces.

Sealed classes and `when` expressions

Using `when` as an expression must return some value, so it must be exhaustive. In most cases, the only way to achieve this is to specify an `else` clause.

```
fun commentValue(value: String) = when {
    value.isEmpty() -> "Should not be empty"
    value.length < 5 -> "Too short"
    else -> "Correct"
}

fun main() {
    println(commentValue("")) // Should not be empty
    println(commentValue("ABC")) // Too short
    println(commentValue("ABCDEF")) // Correct
}
```

⁶¹You could still declare an abstract class or an interface as a part of a sealed hierarchy that the client would be able to inherit from another module.

However, there are also cases in which Kotlin knows that we have specified all possible values. For example, when we use a `when`-expression with an enum value and we compare this value to all possible enum values.

```
enum class PaymentType {
    CASH,
    CARD,
    CHECK
}

fun commentDecision(type: PaymentType) = when (type) {
    PaymentType.CASH -> "I will pay with cash"
    PaymentType.CARD -> "I will pay with card"
    PaymentType.CHECK -> "I will pay with check"
}
```

The power of having a finite set of types as an argument makes it possible to have an exhaustive `when` with a branch for every possible value. In the case of sealed classes or interfaces, this means having `is` checks for all possible subtypes.

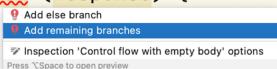
```
sealed class Response<out V>
class Success<V>(val value: V) : Response<V>()
class Failure(val error: Throwable) : Response<Nothing>()

fun handle(response: Response<String>) {
    val text = when (response) {
        is Success -> "Success with ${response.value}"
        is Failure -> "Error"
        // else is not needed here
    }
    print(text)
}
```

Also, IntelliJ automatically suggests adding the remaining branches. This makes sealed classes very convenient when we need to cover all possible types.

```
sealed class Result<out V>
data class Success<V>(val value: V) : Result<V>()
data class Failure(val error: Throwable) : Result<Nothing>()

fun handle(response: Result<String>) {
    val text = when (response) {
        }
```



The screenshot shows an IDE interface with a code editor containing the above Kotlin code. A mouse cursor is hovering over the word 'when'. A context menu is open, displaying three options: 'Add else branch' (with a red exclamation mark icon), 'Add remaining branches' (with a blue question mark icon), and 'Inspection 'Control flow with empty body'' (with a gear icon). Below the menu, a note says 'Press ⌘Space to open preview'.

Note that when an `else` clause is not used and we add another subclass of this sealed class, the usage needs to be adjusted by including this new type. This is convenient in local code as it forces us to handle this new class in exhaustive `when` expressions. The inconvenient part is that when this sealed class is part of the public API of a library or shared module, adding a subtype is a breaking change because all modules that use exhaustive `when` need to cover one more possible type.

Sealed vs enum

Enum classes are used to represent a set of values. Sealed classes or interfaces represent a set of subtypes that can be made with classes or object declarations. This is a significant difference. A class is more than a value. It can have many instances and can be a data holder. Think of `Response`: if it were an enum class, it couldn't hold `value` or `error`. Sealed subclasses can each store different data, whereas an enum is just a set of values.

Use-cases

We use sealed classes whenever we want to express that there is a concrete number of subclasses of a class.

```

sealed class MathOperation
class Plus(val left: Int, val right: Int) : MathOperation()
class Minus(val left: Int, val right: Int) : MathOperation()
class Times(val left: Int, val right: Int) : MathOperation()
class Divide(val left: Int, val right: Int) : MathOperation()

sealed interface Tree
class Leaf(val value: Any?) : Tree
class Node(val left: Tree, val right: Tree) : Tree

sealed interface Either<out L, out R>
class Left<out L>(val value: L) : Either<L, Nothing>
class Right<out R>(val value: R) : Either<Nothing, R>

sealed interface AdView
object FacebookAd : AdView
object GoogleAd : AdView
class OwnAd(val text: String, val imgUrl: String) : AdView

```

The key benefit is that when-expression can easily cover all possible types in a hierarchy using is-checks. A when-condition with a sealed element as a value ensures the compiler performs exhaustive type checking, and our program can only represent valid states.

```

fun BinaryTree.height(): Int = when (this) {
    is Leaf -> 1
    is Node -> maxOf(this.left.height(), this.right.height())
}

```

However, expressing that a hierarchy is restricted improves readability. Finally, when we use the `sealed` modifier, we can use reflection to find all the subclasses⁶²:

⁶²This requires the `kotlin-reflect` dependency. More about reflection in Advanced Kotlin.

```
sealed interface Parent
class A : Parent
class B : Parent
class C : Parent

fun main() {
    println(Parent::class.sealedSubclasses)
    // [class A, class B, class C]
}
```

Summary

Sealed classes and interfaces should be used to represent restricted hierarchies. The when-statement makes it easier to handle each possible sealed subtype and, as a result, to add new methods to sealed elements using extension functions. Abstract classes leave space for new classes to join this hierarchy. If we want to control what the subclasses of a class are, we should use the `sealed` modifier.

Next, we will talk about the last special kind of class that is used to add extra information about our code elements: annotations.

Annotation classes

Another special kind of class in Kotlin is annotations, which we use to provide additional information about an element. Here is an example class that uses the `JvmField`, `JvmStatic`, and `Throws` annotations⁶³.

```
import java.math.BigDecimal
import java.math.MathContext

class Money(
    val amount: BigDecimal,
    val currency: String,
) {
    @Throws(IllegalArgumentException::class)
    operator fun plus(other: Money): Money {
        require(currency == other.currency)
        return Money(amount + other.amount, currency)
    }

    companion object {
        @JvmField
        val MATH = MathContext(2)

        @JvmStatic
        fun eur(amount: Double) =
            Money(amount.toBigDecimal(MATH), "EUR")

        @JvmStatic
        fun usd(amount: Double) =
            Money(amount.toBigDecimal(MATH), "USD")

        @JvmStatic
        fun pln(amount: Double) =
    }
}
```

⁶³The `JvmField`, `JvmStatic`, and `Throws` annotations are described in the book *Advanced Kotlin* and are used to customize how Kotlin elements can be used in Java code.

```
        Money(amount.toBigDecimal(MATH), "PLN")
    }
}
```

You can also define your own annotation. This is an example of annotation declaration and usage:

```
annotation class Factory
annotation class FactoryFunction(val name: String)

@Factory
class CarFactory {

    @FactoryFunction(name = "toyota")
    fun makeToyota(): Car = Toyota()

    @FactoryFunction(name = "skoda")
    fun makeSkoda(): Car = Skoda()
}

abstract class Car
class Toyota : Car()
class Skoda : Car()
```

You might be asking yourself what these annotations do. The answer is surprisingly simple: absolutely nothing. Annotations, by themselves, are not active and do not change how our code works. They only hold information. However, many libraries depend on annotations and behave according to what we specify with them.

Many important libraries use a mechanism called *annotation processing*. How it works is simple: there are classes called *annotation processors* that are running when we build our code. They analyze our code and generate extra code. They generally strongly depend on annotations. This new code is not part of our project sources, but we can access it once it has been generated. This fact is used by libraries that use annotation processing. So, take a look at this class, which uses the Java `Mockito` library with an annotation processor:

```
class DoctorServiceTest {  
  
    @Mock  
    lateinit var doctorRepository: DoctorRepository  
  
    lateinit var doctorService: DoctorService  
  
    @Before  
    fun init() {  
        MockitoAnnotations.initMocks(this)  
        doctorService = DoctorService(doctorRepository)  
    }  
  
    // ...  
}
```

The `doctorRepository` property is annotated as `Mock`, which is interpreted by the Mock processor so that this variable can get a mock value. This processor generates a class that creates and sets a value for the `doctorRepository` property in `DoctorServiceTest`. Of course, this generated class will not work by itself as it needs to be started. This is what `MockitoAnnotations.initMocks(this)` is for: it uses reflection to call this generated class.

Annotation processing is better described in *Advanced Kotlin*, in the *Annotation Processing* chapter.

Meta-annotations

Annotations that are used to annotate other annotations are known as meta-annotations. There are four key meta-annotations from Kotlin stdlib:

- `Target` indicates the kinds of code elements that are possible targets of an annotation. As arguments, it accepts `AnnotationTarget` enum values, which include values like `CLASS`, `PROPERTY`, `FUNCTION`, etc.

- `Retention` determines whether an annotation is stored in the binary output of compilation and is visible for reflection. By default, both are true.
- `Repeatable` determines that an annotation is applicable twice or more in a single code element.
- `MustBeDocumented` determines that an annotation is part of a public API and should therefore be included in the generated documentation for the element to which the annotation is applied.

Here are example usages of some of these annotations:

```
@MustBeDocumented
@Target(AnnotationTarget.CLASS)
annotation class Factory

@Repeatable
@Target(AnnotationTarget.FUNCTION)
annotation class FactoryFunction(val name: String)
```

Annotating primary constructor

To annotate the primary constructor, it is necessary to use the `constructor` keyword as part of its definition, before the parentheses.

```
// JvmOverloads annotates primary constructor
class User @JvmOverloads constructor(
    val name: String,
    val surname: String,
    val age: Int = -1,
)
```

List literals

When we specify an annotation with an array value, we can use a special syntax called “array literal”. This means that instead of using `arrayof`, we can declare an array using square brackets.

```
annotation class AnnotationWithList(  
    val elements: Array<String>  
)  
  
@AnnotationWithList(["A", "B", "C"])  
val str1 = "ABC"  
  
@AnnotationWithList(elements = ["D", "E"])  
val str2 = "ABC"  
  
@AnnotationWithList(arrayOf("F", "G"))  
val str3 = "ABC"
```

This notation is only allowed for annotations and does not work for defining arrays in any other context in our code.

Summary

Annotations are used to describe our code. They might be interpreted by annotation processors or by classes using run-time reflection. Tools and libraries use this to automate some actions for us. Annotations by themselves are a simple feature, but the possibilities offered by them are amazing⁶⁴.

Let's now move on to a famous Kotlin feature that gives us the ability to extend any type with methods or properties: extensions.

⁶⁴In the book *Advanced Kotlin*, we will see how annotation processors can be implemented and what we can do with them.

Extensions

The most intuitive way to define methods and properties is inside classes. Such elements are called **class members** or, more concretely, **member functions** and **member properties**.

```
class Telephone(
    // member property
    val number: String
) {
    // member function
    fun call() {
        print("Calling $number")
    }
}

fun main() {
    // Usage
    val telephone = Telephone("123456789")
    println(telephone.number) // 123456789
    telephone.call() // Calling 123456789
}
```

On the other hand, Kotlin allows another way to define functions and properties that are called on an instance: extensions. **Extension functions** are defined like regular functions, but they additionally have an extra type (and dot) before the function name. In the example below, the `call` function is defined as an extension function on `Telephone`, so it needs to be called on an instance of this type.

```
class Telephone(  
    val number: String  
)  
  
fun Telephone.call() {  
    print("Calling $number")  
}  
  
fun main() {  
    // Usage  
    val telephone = Telephone("123456789")  
    telephone.call() // Calling 123456789  
}
```

Both member functions and extension functions are referred to as methods.

Extension functions can be defined on types we don't control, for instance `String`. This gives us the power to extend external APIs with our own functions.

```
fun String.remove(value: String) = this.replace(value, "")  
  
fun main() {  
    println("Who Framed Roger Rabbit?".remove(" "))  
    // WhoFramedRogerRabbit?  
}
```

Take a look at the example above. We defined the extension function `remove` on `String`, so we need to call this function on an object of type `String`. Inside the function, we reference this object using the `this` keyword, just like inside member functions. The `this` keyword can also be used implicitly.

```
// explicit this
fun String.remove(value: String) = this.replace(value, "")

// implicit this
fun String.remove(value: String) = replace(value, "")
```

The `this` keyword is known as the **receiver**. Inside extension functions, we call it an **extension receiver**. Inside member functions, we call it a **dispatch receiver**. The type we extend with the extension function is called the **receiver type**.

The diagram shows a code snippet for an extension function:

```
Receiver type
  ↗
  fun String.remove(value: String): String {
    return this.replace(value, newValue: "")}
  ↙
  Receiver
  (extension receiver in this case)
```

A curly brace above the function body is labeled "Receiver type". A curly brace below the `this.replace` line is labeled "Receiver" with the subtitle "(extension receiver in this case)".

Extension functions behave a lot like member functions. When developers learn this, they are often concerned about objects' safety, but this isn't a problem as extensions do not have any special access to class elements. The only difference between top-level extension functions and other top-level functions is that they are called on an instance instead of receiving this instance as a regular argument. To see this more clearly, let's take a look under the hood of extension functions.

Extension functions under the hood

To understand extension functions, let's again use “Tools > Kotlin > Show Kotlin bytecode” and “Decompile” (as explained in chapter *Your first program in Kotlin* in section *What*

is under the hood on JVM?). We will compile and decompile to Java our `remove` function definition and its call:

```
fun String.remove(value: String) = this.replace(value, "")  
  
fun main() {  
    println("A B C".remove(" ")) // ABC  
}
```

As a result, you should see the following code:

```
public final class PlaygroundKt {  
    @NotNull  
    public static final String remove(  
        @NotNull String $this$remove,  
        @NotNull String value  
    ) {  
        // parameters not-null checks  
        return StringsKt.replace$default(  
            $this$remove,  
            value,  
            ""  
            // plus default values  
        );  
    }  
  
    public static final void main(@NotNull String[] args) {  
        // parameter not-null check  
        String var1 = remove("A B C", " ");  
        System.out.println(var1);  
    }  
}
```

Notice what happened to the receiver type: it became a parameter. You can also see that, under the hood, `remove` is not called on an object. It is just a regular static function.

When you define an extension function, you do not really add anything to a class. It is just syntactic sugar. Let's compare the two following implementations of `remove`.

```
fun remove(text: String, value: String) =  
    text.replace(value, "")  
  
fun String.remove(value: String) =  
    this.replace(value, "")
```

Under the hood, they are nearly identical. The difference is in how Kotlin expects them to be called. Regular functions receive all their arguments in regular argument positions. Extension functions are called “on” a value.

Extension properties

An extension cannot hold a state, so it cannot have fields. Although properties do not need fields, they can be defined by their getters and setters. This is why we can define extension properties if they do not need a backing field and are defined by accessors.

```
val <T> List<T>.lastIndex: Int  
    get() = size - 1
```

Extension properties are very popular on Android, where accessing different services is complex and repetitive. Defining extension properties lets us do this much more easily.

```
val Context.inflater: LayoutInflator  
    get() = getSystemService(Context.LAYOUT_INFLATER_SERVICE)  
        as LayoutInflator  
  
val Context.notificationManager: NotificationManager  
    get() = getSystemService(Context.NOTIFICATION_SERVICE)  
        as NotificationManager  
  
val Context.alarmManager: AlarmManager  
    get() = getSystemService(Context.ALARM_SERVICE)  
        as AlarmManager
```

Extension properties can define both a getter and a setter. Here is an extension property that provides a different representation of a user birthdate:

```
class User {  
    // ...  
    var birthdateMillis: Long? = null  
}  
  
var User.birthdate: Date?  
    get() = birthdateMillis?.let(::Date)  
    set(value) {  
        birthdateMillis = value?.time  
    }
```

Extensions vs members

The biggest difference between members and extensions in terms of use is that **extensions need to be imported separately**. For this reason, they can be located in a different package. This fact is used when we cannot add a member ourselves. It is also used in projects designed to separate data and behavior. Properties with fields need to be located in a class, but methods can be located separately as long as they only access the public API of the class.

Thanks to the fact that extensions need to be imported, we can have many extensions with the same name for the same type. This is good because different libraries can provide extra methods without causing a conflict. On the other hand, it would be dangerous to have two extensions with the same name but different behaviors. If you have such a situation, it is a code smell and is a clue that someone has abused the extension function capability.

Another significant difference is that **extensions are not virtual**, meaning that they cannot be redefined in derived classes. This is why if you have an extension defined on both a supertype and a subtype, the compiler decides which

function is chosen based on how the variable is typed, not what its actual class is.

```
open class View
class Button : View()

fun View.printMe() {
    println("I'm a View")
}

fun Button.printMe() {
    println("I'm a Button")
}

fun main() {
    val button: Button = Button()
    button.printMe() // I'm a Button
    val view: View = button
    view.printMe() // I'm a View
}
```

The behavior of extension functions is different from member functions. Member functions are virtual, so up-casting the type of an object does not influence which member function is chosen.

```
open class View {
    open fun printMe() {
        println("I'm a View")
    }
}
class Button: View() {
    override fun printMe() {
        println("I'm a Button")
    }
}

fun main() {
```

```
val button: Button = Button()
button.printMe() // I'm a Button
val view: View = button
view.printMe() // I'm a Button
}
```

This behavior is the result of the fact that extension functions are compiled under the hood into normal functions in which the extension's receiver is placed as the first argument:

```
open class View
class Button : View()

fun printMe(view: View) {
    println("I'm a View")
}

fun printMe(button: Button) {
    println("I'm a Button")
}

fun main() {
    val button: Button = Button()
    printMe(button) // I'm a Button
    val view: View = button
    printMe(view) // I'm a View
}
```

Another consequence of what extensions are under the hood is that **we define extensions on types, not on classes**. This gives us more freedom. For instance, we can define an extension on a nullable or generic type:

```
inline fun CharSequence?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }

    return this == null || this.isBlank()
}

fun Iterable<Int>.sum(): Int {
    var sum: Int = 0
    for (element in this) {
        sum += element
    }
    return sum
}
```

The last important difference is that **extensions are not listed as members in the class reference**. This is why they are not considered by annotation processors; it is also why, when we process a class using annotation processing, we cannot extract elements that should be processed into extensions. On the other hand, if we extract non-essential elements into extensions, we don't need to worry about them being seen by those processors. We don't need to hide them because they are not in the class they extend anyway.

Extension functions on object declarations

We can define extensions on object declarations.

```
object A

fun A.foo() {}

fun main() {
    A.foo()

    val a: A = A
    a.foo()
}
```

To define an extension function on a companion object, we need to use the companion object's real name. If this name is not set explicitly, the default one is "Companion". To define an extension function on a companion object, this companion object must exist. This is why some classes define companion objects without bodies.

```
class A {
    companion object
}

fun A.Companion.foo() {}

fun main() {
    A.foo()

    val a: A.Companion = A
    a.foo()
}
```

Member extension functions

It is possible to define extension functions inside classes. Such functions are known as **member extension functions**.

```
class Telephone {  
    fun String.call() {  
        // ...  
    }  
}
```

Member extension functions are considered a code smell, and we should avoid using them if we don't have a good reason. For a deeper explanation, see [Effective Kotlin, Item 46: Avoid member extensions.](#)

Use-cases

The most important use-case for extensions is adding methods and properties to APIs that we don't control. A good example is displaying a toast or hiding a view on Android. Both these operations are unnecessarily complicated, so we like to define extensions to simplify them.

```
fun Context.toast(message: String) {  
    Toast.makeText(this, message, Toast.LENGTH_LONG).show()  
}  
  
fun View.hide() {  
    this.visibility = View.GONE  
}
```

However, there are also cases where we prefer to use extensions instead of members. Consider the `Iterable` interface, which has only one member function, `iterator`; however, it has many methods, which are defined in the standard library as extensions⁶⁵, like `onEach` or `joinToString`. The fact that these

⁶⁵Roman Elizarov (Project Lead for the Kotlin Programming Language) refers to this as an extension-oriented design in the standard library. Source: <https://elizarov.medium.com/extension-oriented-design-13f4f27deae>

are defined as extensions allows for smaller and more concise interfaces.

```
interface Iterable<out T> {
    operator fun iterator(): Iterator<T>
}
```

```
fun consume(i: Iterable<Int>) {
    i.l
}
    ↪ toString() String
    ↪ spliterator() Spliterator<Int>
    ↪ hashCode() Int
    ↪ equals(other: Any?) Boolean
    ↪ iterator() Iterator<Int>
    ↪ forEach(action: Consumer<in Int!>) Unit
    ↪ forEach {..} (action: ((Int!) -> Unit)!) Unit
    ↪ to(that: B) for A in kotlin Pair<Iterable<Int>, B>
    ↪ map {...} (transform: (Int) -> R) for Iterable<T> in kotlin.collections List<R>
    ↪ joinToString {...} (... transform: ((Int) -> CharSequence)? = ...) for I... String
    ↪ joinTo(buffer: A, separator: CharSequence = ..., prefix: CharSequence = ..., p... A
    ↪ joinToString(separator: CharSequence = ..., prefix: CharSequence = ..., p... String
    ↪ find {...} (predicate: (Int) -> Boolean) for Iterable<T> in kotlin.collect... Int?
    ↪ asIterable() for Iterable<T> in kotlin.collections Iterable<Int>
    ↪ forEachIndexed(action: (Int, Int) -> Unit) for Iterable<T> in kotlin.collect... Unit
    ↪ forEach {...} (action: (Int) -> Unit) for Iterable<T> in kotlin.collections Unit
    ↪ forEachIndexed { index, Int -> ... } (action: (Int, Int) -> Unit) for Iterable... Unit
    ↪ withIndex() for Iterable<T> in kotlin.collections Iterable<IndexedValue<Int>>
    ↪ toList() for Iterable<T> in kotlin.collections List<Int>
    ↪ asSequence() for Iterable<T> in kotlin.collections Sequence<Int>
    ↪ sumOf {...} (selector: (Int) -> Int) for Iterable<T> in kotlin.collections Int
    ↪ sum() for Iterable<Int> in kotlin.collections Int
    ↪ sumOf {...} (selector: (Int) -> Long) for Iterable<T> in kotlin.collections Long
    ↪ sumOf {...} (selector: (Int) -> UInt) for Iterable<T> in kotlin.collections UInt
    ↪ sumOf {...} (selector: (Int) -> ULong) for Iterable<T> in kotlin.collectio... ULong
    ↪ sumOf {...} (selector: (Int) -> Double) for Iterable<T> in kotlin.collect... Double
    ↪ sumOf {...} (selector: (Int) -> BigDecimal) for Iterable<T> in kotlin... BigDecimal
    ↪ sumOf {...} (selector: (Int) -> BigInteger) for Iterable<T> in kotlin... BigInteger
    ↪ distinctBy {...} (selector: (Int) -> K) for Iterable<T> in kotlin.collect... List<Int>
    ↪ distinct() for Iterable<T> in kotlin.collections List<Int>
    ↪ toSet() for Iterable<T> in kotlin.collections Set<Int>
    ↪ flatMap {...} (transform: (Int) -> Iterable<R>) for Iterable<T> in kotlin... List<R>
    ↪ flatMap {...} (transform: (Int) -> Sequence<R>) for Iterable<T> in kotlin... List<R>
    ↪ sortedWith(comparator: Comparator<in Int> /* = Comparator<in Int> */ f... List<Int>
    ↪ ...
}

Press ⌘ to insert, ⌘ to replace Next Tip
```

Extension functions are also more elastic than regular functions. This is mainly because they are defined on types, so we can define extensions on types like `Iterable<Int>` or `Iterable<T>`.

```
fun <T : Comparable<T>> Iterable<T>.sorted(): List<T> {
    if (this is Collection) {
        if (size <= 1) return this.toList()
        @Suppress("UNCHECKED_CAST")
        return (toTypedArray<Comparable<T>>() as Array<T>)
            .apply { sort() }
            .asList()
    }
    return toMutableList().apply { sort() }
}

fun Iterable<Int>.sum(): Int {
    var sum: Int = 0
    for (element in this) {
        sum += element
    }
    return sum
}
```

In bigger projects, we often have similar classes for different parts of our application. Let's say that you implement a backend for an online shop, and you have a class `Product` to represent all the products.

```
import java.math.BigDecimal

class Product(
    val id: String,
    val title: String,
    val imgSrc: String,
    val description: String,
    val price: BigDecimal,
    val type: ProductType,
    // ...
)
```

You also have a similar (but not identical) class called `ProductJson`, which is used to represent the objects you

use in your application API responses or that you read from API requests.

```
class ProductJson(  
    val id: String,  
    val title: String,  
    val img: String,  
    val desc: String,  
    val price: String,  
    val type: String,  
    // ...  
)
```

Instances of `Product` are used in your application, and instances of `ProductJson` are used in the API. These objects need to be separated because, for instance, you don't want to change your API response when you change a property name in an internal class. Yet, we often need to transform between `Product` and `ProductJson`. For this, we could define a member function `toProduct`.

```
class ProductJson(  
    val id: String,  
    val title: String,  
    val img: String,  
    val desc: String,  
    val price: String,  
    val type: String,  
    // ...  
) {  
    fun toProduct() = Product(  
        id = this.id,  
        title = this.title,  
        imgSrc = this.img,  
        description = this.desc,  
        price = BigDecimal(price),  
        type = enumValueOf<ProductType>(this.type)  
    )  
}
```

Not everyone likes this solution as it makes `ProductJson` bigger and more complicated. It is also not useful in transforming from `Product` to `ProductJson` because in most modern architectures we don't want domain classes (like `Product`) to be aware of details such as their API representation. A better solution is to define both `toProduct` and `toProductJson` as extension functions, then locate them together next to the `ProductJson` class. It is good to locate those transformation functions next to each other, because they have a lot in common.

```
class ProductJson(  
    val id: String,  
    val title: String,  
    val img: String,  
    val desc: String,  
    val price: String,  
    val type: String,  
    // ...  
)  
  
fun ProductJson.toProduct() = Product(  
    id = this.id,  
    title = this.title,  
    imgSrc = this.img,  
    description = this.desc,  
    price = BigDecimal(this.price),  
    type = enumValueOf<ProductType>(this.type)  
)  
  
fun Product.toJson() = ProductJson(  
    id = this.id,  
    title = this.title,  
    img = this.imgSrc,  
    desc = this.description,  
    price = this.price.toString(),  
    type = this.type.name  
)
```

This seems to be a popular pattern, both on the backend and

in Android applications.

Summary

In this chapter, we've learned about extensions - a powerful Kotlin feature that is often used to create convenient and meaningful utils and to control our code better. However, with great power comes great responsibility. We should not be worried about using extensions, but we should use them consciously and only where they make sense.

In the next chapter, we will finally introduce collections so that we can talk about lists, sets, maps, and arrays. There's a lot ahead, so get ready.

Collections

Collections are one of the most important concepts in programming. They are types that represent groups of elements. In Kotlin, the most important collection types are:

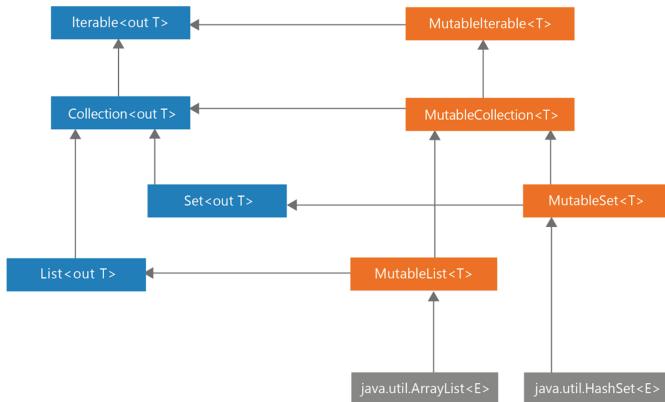
- `List`, which represents an ordered collection of elements. The same elements can occur multiple times. A list's elements can be accessed by indices (zero-based integer numbers that reflect elements' positions). An example might be a list of songs in a queue: the order of the songs is important, and each song can appear in multiple places.
- `Set`, which represents a collection of unique elements. It reflects the mathematical abstraction of a set: a group of objects without repetitions. Sets might not respect element order (however, the default set used by Kotlin does respect element order). An example might be a set of winning numbers in a lottery: they must be unique, but their order does not matter.
- `Map` (known as a dictionary in some other languages), which represents a set of key-value pairs. Keys must be unique, and each of them points to exactly one value. Multiple keys can be associated with the same values. Maps are useful for expressing logical connections between elements.

There are also arrays, which are typically considered a low-level primitive used by other collections under the hood.

In this chapter, we will cover the most important topics regarding collections, starting from how they are organized, how they are created, special kinds of collections, and how all these kinds of collections are used in practice. This is a long chapter, so let's get started.

The hierarchy of interfaces

In Kotlin, a whole hierarchy of interfaces is used to represent different kinds of collections. Take a look at the diagram below.



Relations between interfaces representing collections. Blue elements are read-only. Orange elements are mutable. Classes like `ArrayList` or `HashSet` implement mutable variants but can be up-casted to read-only.

At the top of the hierarchy, there is `Iterable`, which represents a sequence of elements that can be iterated over. We can iterate over `Iterable` objects using a for-loop thanks to its `iterator` method.

```
interface Iterable<out T> {
    operator fun iterator(): Iterator<T>
}
```

The next type is `Collection`, which represents a collection of elements. Its methods are read-only (no methods for manipulating the elements are available), so this interface does not allow any modifications.

```
interface Collection<out E> : Iterable<E> {
    val size: Int
    fun isEmpty(): Boolean
    operator fun contains(element: E): Boolean
    override fun iterator(): Iterator<E>
    fun containsAll(elements: Collection<E>): Boolean
}
```

Notice that only `List` and `Set` are subtypes of `Collection` and `Iterable`. `Map` and arrays are not part of this hierarchy; however, we can also iterate over them using a for-loop.

All the interfaces described so far are read-only, so they have methods that allow what is inside to be read (like `get`, `contains`, `size`) but not modified. `MutableIterable`, `MutableCollection` and `MutableList` are sub-interfaces of, respectively, `Iterable`, `Collection` and `List`, which add methods for modifying elements (like `remove`, `clear`, `add`).

The actual classes used when we operate on collections are platform-specific. For instance, if you create a list using the `listOf("A", "B")` function on Kotlin/JVM version 1.7.20, the actual class is `Arrays.ArrayList` from the Java Standard Library; however, if you use Kotlin/JS version 1.7.20, the actual class is a JavaScript array. The point is to expect not a concrete class but an object that represents a list (and therefore implements a `List` interface). This is an application of a general idea: use abstractions to make it easier to change the underlying representations (e.g., for performance reasons) without pushing breaking changes⁶⁶.

Mutable vs read-only types

The distinction between mutable and read-only interfaces is very important. For instance, the `listOf` function returns `List`, which represents a read-only collection. `List` does not have any functions that would allow its modification (functions

⁶⁶This topic is covered in *Effective Kotlin*, especially in Item 27: *Use abstraction to protect code against changes*.

like `add` or `remove`). This means a collection object cannot mutate, but this doesn't mean we cannot update a variable that contains a collection.

It's a similar story with `Int` or `String`. Both are immutable, so they cannot change internally; however, we update their values with operators like plus.

```
fun main() {
    var a = 100
    a = a + 10
    a += 1
    println(a) // 111

    var str = "A"
    str = str + "B"
    str += "C"
    println(str) // ABC
}
```

The same goes for read-only collections: we can use operators to create a new collection with an updated value.

```
fun main() {
    var list = listOf("A", "B")
    list = list + "C"
    println(list) // [A, B, C]
    list = list + listOf("D", "E")
    println(list) // [A, B, C, D, E]
    list = listOf("Z") + list
    println(list) // [Z, A, B, C, D, E]
}
```

In contrast to read-only lists, mutable lists can be modified internally. So, if you create a collection using the `mutableListOf` function, the result object is `MutableList`, which supports operations like `add`, `clear` or `remove`.

```
fun main() {
    val mutable = mutableListOf("A", "B")
    mutable.add("C")
    mutable.remove("A")
    println(mutable) // [B, C]
}
```

You can easily transform between a mutable and a read-only list with `toList` and `toMutableList`. However, you often do not need to explicitly transform from a `MutableList` to a `List`. `MutableList` is a subtype of `List`, so a `MutableList` can be used as a `List`.

In the upcoming sections, we will see the most important operators for modifying read-only collections, and methods that can be used to modify mutable collections.

Creating collections

Most languages have support for a feature called a “collection literal”, which is special syntax for creating a certain collection type based on the provided list of elements.

```
// JavaScript
const arr = ["A", "B"] // an array of strings
// Python
numbers = [1, 2, 3] // a list of numbers
names = {"Alex", "Barbara"} // a set of strings
```

In Kotlin, this role is performed by top-level functions. By convention, their names start with the name of the type they produce (starting from lower case) and the `of` suffix. Here are a few examples.

```
fun main() {
    // We create `List` using `listOf` function.
    val list: List<String> = listOf("A", "B", "C")
    // We create `MutableList` using `mutableListOf` function.
    val mutableList: MutableList<Int> = mutableListOf(1, 2, 3)

    // We create `Set` using `setOf` function.
    val set: Set<Double> = setOf(3.14, 7.11)
    // We create `MutableSet` using `mutableSetOf` function.
    val mutableSet: MutableSet<Char> = mutableSetOf('A', 'B')

    // We create `Map` using `mapOf` function.
    val map: Map<Char, String> =
        mapOf('A' to "Alex", 'B' to "Ben")
    // We create `MutableMap` using `mutableMapOf` function.
    val mutableMap: MutableMap<Int, Char> =
        mutableMapOf(1 to 'A', 2 to 'B')

    // We create `Array` using `arrayOf` function.
    val array: Array<String> = arrayOf("Dukaj", "Sapkowski")
    // We create `IntArray` using `intArrayOf` function.
    val intArray: IntArray = intArrayOf(9, 8, 7)

    // We create `ArrayList` using `arrayListOf` function.
    val arrayList: ArrayList<String> = arrayListOf("M", "N")
}
```

To all these classes, we provide initial elements as arguments. The only exception is map, which is a set of key-value pairs, so we specify the initial pairs using `Pair`, which we typically create using the `to` function (as explained in the chapter Data classes).

We can also transform one collection into another. This can often be done using a method whose name is the type we want to achieve, preceded by the `to` prefix.

```
fun main() {
    val list: List<Char> = listOf('A', 'B', 'C')
    val mutableListOf: MutableList<Char> = list.toMutableList()
    val set: Set<Char> = mutableListOf.toSet()
    val mutableSet: MutableSet<Char> = set.toMutableList()
    val array: Array<Char> = mutableSet.toTypedArray()
    val charArray: CharArray = array.toCharArray()
    val list2: List<Char> = charArray.toList()
}
```

Lists

List is the most basic type of collection. You can treat it as the default collection type. It represents an ordered list of elements.

```
fun main() {
    val list = listOf("A", "B", "C")
    println(list) // [A, B, C]
}
```

List is a generic class. The result type of `listOf` is `List<T>`, where `T` is the type of the elements in this list. Since we have a list with string values in the code above, the type is `List<String>`. More about generic classes in the chapter **Generics**.

```
fun main() {
    val list: List<String> = listOf("A", "B", "C")
    println(list) // [A, B, C]
    val ints: List<Int> = listOf(1, 2, 3)
    println(ints) // [1, 2, 3]
}
```

Modifying lists

When you need to modify the elements of a list, you have two options:

1. Use a read-only list in a `var` variable, and modify it using operators like plus or minus.
2. Use a mutable list in a `val` variable, and modify it using `MutableList` methods like `add`, `addAll` or `remove`.

```
fun main() {  
    var list = listOf("A", "B")  
    list = list + "C"  
    println(list) // [A, B, C]  
    list = list + listOf("D", "E")  
    println(list) // [A, B, C, D, E]  
    list = listOf("Z") + list  
    println(list) // [Z, A, B, C, D, E]  
    list = list - "A"  
    println(list) // [Z, B, C, D, E]  
  
    val mutable = mutableListOf("A", "B")  
    mutable.add("C")  
    println(mutable) // [A, B, C]  
    mutable.addAll(listOf("D", "E"))  
    println(mutable) // [A, B, C, D, E]  
    mutable.add(0, "Z") // The first number is index  
    println(mutable) // [Z, A, B, C, D, E]  
    mutable.remove("A")  
    println(mutable) // [Z, B, C, D, E]  
}
```

Since the beginning of Kotlin, there have been discussions about which of these two approaches should be preferred. The first gives more freedom⁶⁷, but the second is considered more efficient⁶⁸.

You can also use the `+=` operator to add an element or a collection to a `var` variable that points to a read-only list, or to a `var` variable that points to a mutable list.

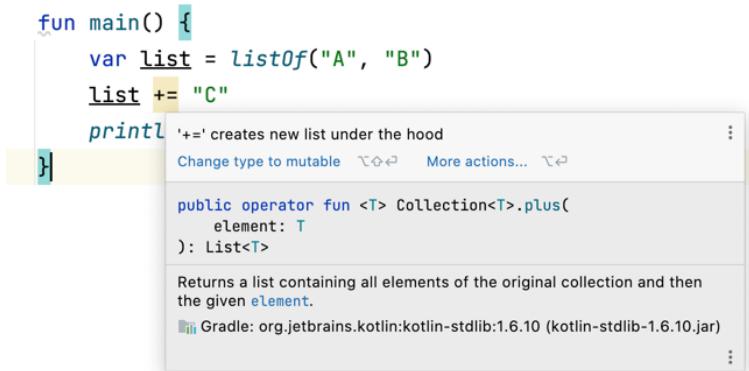
⁶⁷As presented in Effective Kotlin, Item 1: Limit mutability.

⁶⁸As presented in Effective Kotlin, Item 56: Consider using mutable collections.

```
fun main() {
    var list = listOf("A", "B")
    list += "C"
    println(list) // [A, B, C]

    val mutable = mutableListOf("A", "B")
    mutable += "C"
    println(mutable) // [A, B, C]
}
```

However, using `+=` for read-only lists results in a warning that a new collection has been created under the hood, which can lead to performance issues when we are dealing with large lists.



Checking a list's size or if it is empty

You can get the number of elements in a list using the `size` property.

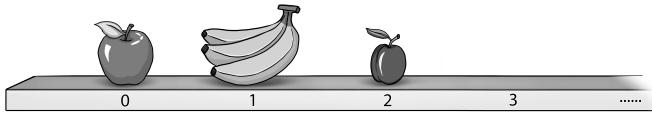
```
fun main() {
    val list = listOf("A", "B", "C")
    println(list.size) // 3
}
```

A list is considered empty when its size is 0. You can also check this using the `isEmpty` method.

```
fun main() {  
    val list = listOf("A", "B", "C")  
    println(list.size == 0) // false  
    println(list.isEmpty()) // false  
  
    val empty: Set<Int> = setOf()  
    println(empty.size == 0) // true  
    println(empty.isEmpty()) // true  
}
```

Lists and indices

Lists allow elements to be retrieved by their index, which is a number that represents the element's position. The index of the first element is always 0, and each next element in the list has the next index. You can imagine a list to be like an infinite shelf for items, where there is a label with a number below each item.



To get an element by an index, we use the box bracket. This is a synonym of the `get` method. Both these methods throw an `IndexOutOfBoundsException` when you try to get an element at an index that does not exist.

```
fun main() {
    val list = listOf("A", "B")
    println(list[1]) // B
    println(list.get(1)) // B
    println(list[3]) // Runtime error
}
```

If you are not sure if an index is correct, it is safer to use either `getOrNull`, which returns `null` in the case of an incorrect index, or `getOrElse`, which specifies the default value⁶⁹.

```
fun main() {
    val list = listOf("A", "B")
    println(list.getOrNull(1)) // B
    println(list.getOrElse(1) { "X" }) // B

    println(list.getOrNull(3)) // null
    println(list.getOrElse(3) { "X" }) // X
}
```

You can find the index of an element using the `indexOf` method. It returns `-1` when there is no matching element in the list.

```
fun main() {
    val list = listOf("A", "B")
    println(list.indexOf("A")) // 0
    println(list.indexOf("B")) // 1
    println(list.indexOf("Z")) // -1
}
```

In a mutable list, you can modify an element at a certain index using the box bracket in an assignment, or using the `set` method.

⁶⁹The default value is calculated by a functional argument, such as a lambda expression; we will describe this in the next book *Functional Kotlin*

```
fun main() {
    val mutable = mutableListOf("A", "B", "C")
    mutable[1] = "X"
    println(mutable) // [A, X, C]
    mutable.set(1, "Y")
    println(mutable) // [A, Y, C]
}
```

Checking if a list contains an element

You can check if a list contains an element using the `contains` method or the `in` operator.

```
fun main() {
    val letters = listOf("A", "B", "C")
    println(letters.contains("A")) // true
    println(letters.contains("Z")) // false
    println("A" in letters) // true
    println("Z" in letters) // false
}
```

You can also check whether a collection does not contain an element using the `!in` operator.

```
fun main() {
    val letters = listOf("A", "B", "C")
    println("A" !in letters) // false
    println("Z" !in letters) // true
}
```

Iterating over a list

You can iterate over a list using a for-loop. Just place the list on the right side of `in`.

```
fun main() {
    val letters = listOf("A", "B", "C")
    for (letter in letters) {
        print(letter)
    }
}
```

Since `MutableList` implements `List`, all these operations can also be used on mutable lists.

These are the most basic operations on lists. We will cover more of them in the next book: *Functional Kotlin*.

Sets

We use **sets** instead of lists when:

1. we want to ensure that elements in our collection are unique (sets keep only unique elements),
2. we frequently look for an element in a collection (finding elements in a set is much more efficient than doing so in a list⁷⁰).

Sets are quite similar to lists, which is why similar methods are used to operate on them. However, sets do not treat order as seriously as lists, and some kinds of sets do not respect order at all. This is why we cannot get elements by index.

We create a set using the `setOf` function; then we specify its values using arguments.

⁷⁰The default set is based on a hash table algorithm, which makes finding an element with a properly implemented `hashCode` really fast. This operation time does not depend on the number of elements in the set (so has O(1) complexity). For details about how this hash table algorithm works, see Effective Kotlin Item 43: Respect the contract of `hashCode`.

```
fun main() {  
    val set = setOf('A', 'B', 'C')  
    println(set) // [A, B, C]  
}
```

Set is a generic class. The result type of `setOf` is `Set<T>`, where `T` is the type of elements in this set. Since we have a set with char values in the code above, the type is `Set<Char>`.

```
fun main() {  
    val set: Set<Char> = setOf('A', 'B', 'C')  
    println(set) // [A, B, C]  
    val ints: Set<Long> = setOf(1L, 2L, 3L)  
    println(ints) // [1, 2, 3]  
}
```

Modifying sets

You can add elements to a read-only set in the same way as to a read-only list: using plus or minus.

```
fun main() {  
    var set = setOf("A", "B")  
    set = set + "C"  
    println(set) // [A, B, C]  
    set = set + setOf("D", "E")  
    println(set) // [A, B, C, D, E]  
    set = setOf("Z") + set  
    println(set) // [Z, A, B, C, D, E]  
    set = set - "A"  
    println(set) // [Z, B, C, D, E]  
}
```

You can also use a mutable set and its `add`, `addAll` or `remove` methods.

```
fun main() {
    val mutable = mutableSetOf("A", "B")
    mutable.add("C")
    println(mutable) // [A, B, C]
    mutable.addAll(listOf("D", "E"))
    println(mutable) // [A, B, C, D, E]
    mutable.remove("B")
    println(mutable) // [A, C, D, E]
}
```

Elements in a set are unique

Sets accept only unique elements. If elements repeat during set creation, only the first occurrence will be present in the set.

```
fun main() {
    val set = setOf("A", "B", "C", "B")
    println(set) // [A, B, C]
}
```

Adding an element that is equal to an element already present in a set is ignored.

```
fun main() {
    val set = setOf("A", "B", "C")
    println(set + "D") // [A, B, C, D]
    println(set + "B") // [A, B, C]

    val mutable = mutableSetOf("A", "B", "C")
    mutable.add("D")
    mutable.add("B")
    println(mutable) // [A, B, C, D]
}
```

Two elements are considered different when comparing them using the double equality sign returns `false`.

```
// by default, each object from a regular class is unique
class Cat(val name: String)

// if the data modifier is used,
// two instances with the same properties are equal
data class Dog(val name: String)

fun main() {
    val cat1 = Cat("Garfield")
    val cat2 = Cat("Garfield")
    println(cat1 == cat2) // false
    println(setOf(cat1, cat2)) // [Cat@4eec7777, Cat@3b07d329]

    val dog1 = Dog("Rex")
    val dog2 = Dog("Rex")
    println(dog1 == dog2) // true
    println(setOf(dog1, dog2)) // [Dog(name=Rex)]
}
```

The most efficient way to remove duplicates from a list is by transforming it into a set.

```
fun main() {
    val names = listOf("Jake", "John", "Jake", "James", "Jan")
    println(names) // [Jake, John, Jake, James, Jan]
    val unique = names.toSet()
    println(unique) // [Jake, John, James, Jan]
}
```

Checking a set's size or if it is empty

You can always check the number of elements in a set using the `size` property.

```
fun main() {  
    val set = setOf('A', 'B', 'C')  
    println(set.size) // 3  
}
```

To check if a set is empty, you can compare its size to 0, or you can use the `isEmpty` method.

```
fun main() {  
    val set = setOf('A', 'B', 'C')  
    println(set.size == 0) // false  
    println(set.isEmpty()) // false  
  
    val empty: Set<Int> = setOf()  
    println(empty.size == 0) // true  
    println(empty.isEmpty()) // true  
}
```

Checking if a set contains an element

You can check if a set contains a certain element by using the `contains` method or the `in` operator. Both these options return `true` if there is an element equal to the element you are looking for in the set; otherwise, it returns `false`.

```
fun main() {  
    val letters = setOf('A', 'B', 'C')  
    println(letters.contains('A')) // true  
    println(letters.contains('Z')) // false  
    println('A' in letters) // true  
    println('Z' in letters) // false  
}
```

You can also check whether a set does not contain an element using the `!in` operator.

```
fun main() {
    val letters = setOf("A", "B", "C")
    println("A" !in letters) // false
    println("Z" !in letters) // true
}
```

Iterating over sets

You can iterate over a set using a for-loop. Just place the set on the right side of `in`.

```
fun main() {
    val letters = setOf('A', 'B', 'C')
    for (letter in letters) {
        print(letter)
    }
}
```

Maps

We use maps to keep associations from keys to their values. For instance:

- From user id to an object representing this user.
- From a website to its IP address.
- From a configuration name to data stored in this configuration.

```
class CachedApiArticleRepository(  
    val articleApi: ArticleApi  
) {  
    val articleCache: MutableMap<String, String> =  
        mutableMapOf()  
  
    fun getContent(key: String) =  
        articleCache.getOrPut(key) {  
            articleApi.fetchContent(key)  
        }  
}  
  
class DeliveryMethodsConfiguration(  
    val deliveryMethods: Map<String, DeliveryMethod>  
)  
  
class TokenRepository {  
    private var tokenToUser: Map<String, User> = mapOf()  
  
    fun getUser(token: String) = tokenToUser[token]  
  
    fun addToken(token: String, user: User) {  
        tokenToUser[token] = user  
    }  
}
```

You can create a map using the `mapOf` function and then use key-value pairs as arguments to specify key-value associations. For instance, I might define a map that associates countries with their capitals. Pairs can be defined using a constructor or the `to` function.

```
fun main() {
    val capitals = mapOf(
        "USA" to "Washington DC",
        "Poland" to "Warsaw",
    )
    //    val capitals = mapOf(
    //        Pair("USA", "Washington DC"),
    //        Pair("Poland", "Warsaw"),
    //    )
    println(capitals) // {USA=Washington DC, Poland=Warsaw}
}
```

Map is a generic class. The result type is `Map<K, V>`, where `K` is the key type, and `V` is the value type. In the case of the map from the `capitals` variable above, both the keys and the values are of type `String`, so the map type is `Map<String, String>`. However, a key does not need to be the same type as its value. Consider a map with associations between letters and their positions in the English alphabet, as in the example below. Its type is `Map<Char, Int>` because its keys are characters and its values are integers.

```
fun main() {
    val capitals: Map<String, String> = mapOf(
        "USA" to "Washington DC",
        "Poland" to "Warsaw",
    )
    println(capitals) // {USA=Washington DC, Poland=Warsaw}

    val alphabet: Map<Char, Int> =
        mapOf('A' to 1, 'B' to 2, 'C' to 3)
    println(alphabet) // {A=1, B=2, C=3}
}
```

Finding a value by a key

To find a value by a key, you can use the `get` function or box brackets with the key. For instance, to find the value

for the key 'A' in the `alphabet` map, use `alphabet.get('A')` or `alphabet['A']`. The result has a nullable value type, which is `Int?` in this case. Why nullable? If the key you asked for is not in the map, then `null` will be returned.

```
fun main() {
    val alphabet: Map<Char, Int> =
        mapOf('A' to 1, 'B' to 2, 'C' to 3)
    val number: Int? = alphabet['A']
    println(number) // 1
    println(alphabet['B']) // 2
    println(alphabet['&']) // null
}
```

All the basic maps are optimized to make finding a value by a key a very fast operation⁷¹.

Adding elements to a map

Just like a regular list or a regular set, a regular map is read-only, so it does not have methods that would allow elements to be added or removed. However, you can use the plus sign to create a new map with new entries. If you add a pair to a map, the result is a map with the new entry. If you add two maps together, the result is a merge of these two maps.

⁷¹The default map is based on a hash table algorithm, which makes finding an element by key really fast (when this key has a properly implemented `hashCode` method). This operation time does not depend on the number of entries in the map (so has O(1) complexity). For details about how this hash table algorithm works, see Effective Kotlin Item 43: Respect the contract of `hashCode`.

```
fun main() {
    val map1 = mapOf('A' to "Alex", 'B' to "Bob")
    val map2 = map1 + ('C' to "Celina")
    println(map1) // {A=Alex, B=Bob}
    println(map2) // {A=Alex, B=Bob, C=Celina}
    val map3 = mapOf('D' to "Daniel", 'E' to "Ellen")
    val map4 = map2 + map3
    println(map3) // {D=Daniel, E=Ellen}
    println(map4)
    // {A=Alex, B=Bob, C=Celina, D=Daniel, E=Ellen}
}
```

Beware that duplicate keys are not allowed; so, when you add a new value with an existing key, it replaces the old value.

```
fun main() {
    val map1 = mapOf('A' to "Alex", 'B' to "Bob")
    val map2 = map1 + ('B' to "Barbara")
    println(map1) // {A=Alex, B=Bob}
    println(map2) // {A=Alex, B=Barbara}
}
```

You can also remove a key from a map using the minus sign.

```
fun main() {
    val map1 = mapOf('A' to "Alex", 'B' to "Bob")
    val map2 = map1 - 'B'
    println(map1) // {A=Alex, B=Bob}
    println(map2) // {A=Alex}
}
```

Checking if a map contains a key

You can check if your map contains a key using the `in` keyword or the `containsKey` method.

```
fun main() {  
    val map = mapOf('A' to "Alex", 'B' to "Bob")  
    println('A' in map) // true  
    println(map.containsKey('A')) // true  
    println('Z' in map) // false  
    println(map.containsKey('Z')) // false  
}
```

Checking map size

You can check how many entries you have in a map using the `size` property.

```
fun main() {  
    val map = mapOf('A' to "Alex", 'B' to "Bob")  
    println(map.size) // 2  
}
```

Iterating over maps

You can iterate over a map using a for-loop. You iterate over entries that contain `key` and `value` properties.

```
fun main() {  
    val map = mapOf('A' to "Alex", 'B' to "Bob")  
    for (entry in map) {  
        println("${entry.key} is for ${entry.value}")  
    }  
}  
// A is for Alex  
// B is for Bob
```

You can also destructure each entry into two variables. Kotlin supports destructuring in a for-loop. Take a look at the example below.

```
fun main() {
    val map = mapOf('A' to "Alex", 'B' to "Bob")
    for ((letter, name) in map) {
        println("$letter is for $name")
    }
}
// A is for Alex
// B is for Bob
```

Mutable maps

You can create a mutable map using `mutableMapOf`. The result type is `MutableMap`, which supports methods that modify this object. Using it we can:

- add new entries to the map using the `put` method, or box brackets and assignment,
- remove an entry by key using the `remove` method.

```
fun main() {
    val map: MutableMap<Char, String> =
        mutableMapOf('A' to "Alex", 'B' to "Bob")
    map.put('C', "Celina")
    map['D'] = "Daria"
    println(map) // {A=Alex, B=Bob, D=Daria, C=Celina}
    map.remove('B')
    println(map) // {A=Alex, D=Daria, C=Celina}
}
```

Using arrays in practice

Array is a very basic data structure that strongly relates to how memory is organized. Our computer's memory is like a big parking lot, where each place has a sequential number. An array is like a reservation for a number of adherent spaces. With such a reservation, it is really easy to iterate over the cars you own. It is also easy to find a car with a specific index.

Let's say that an array starts at position 1024 in your memory, and you need to find the element at index 100 in the array. You also know that each element takes 4 positions (an array reserves constant space for its elements, which in most cases is the size of the memory reference). This is an easy problem: our element starts at the position $1024 + 100 * 4 = 1424$. Accessing an element at a certain position is a very simple and efficient operation, which is a big advantage of using arrays.

Using arrays directly is harder than using other kinds of collections. They have a constant size, a limited number of operations, they do not implement any interface, and they do not override the `toString`, `equals` or `hashCode` methods. However, arrays are used by many other data structures under the hood. For instance, when you use `mutableListOf` on Kotlin/JVM, the result object is `ArrayList`, which keeps elements in an array. This is why finding an element at an index in the default list is so efficient. So, `ArrayList` has the advantages of arrays, but it offers much more. Arrays have a constant size, so you cannot add more elements than their size allows. When you add an element to an `ArrayList` and its internal array is full already, it creates a bigger one and fills it with the previous values. We consider lists a preferred option to arrays, and we restrict the usage of arrays to performance-critical parts of our applications⁷².

Arrays are also used by the default `Set` and `Map` that we use in Kotlin. Both are based on a hash table algorithm that needs to use an array to work efficiently.

Nevertheless, let's see how arrays can be used directly. We create an array using the `arrayOf` function. This creates an instance of class `Array` and of type `Array<T>`, where `T` is the type of the elements. To get an element at a certain index, we can use box brackets or the `get` method. To modify an element at a certain position, you can use box brackets or the `set` method. You can also get an array's size using the `size` property or by iterating over the array using a for-loop.

⁷²See Effective Kotlin, Item 55: Consider Arrays with primitives for performance-critical processing.

```
fun main() {
    val arr: Array<String> = arrayOf("A", "B", "C")
    println(arr[0]) // A
    println(arr.get(0)) // A
    println(arr[1]) // B
    arr[1] = "D"
    println(arr[1]) // D
    arr.set(2, "E")
    println(arr[2]) // E
    println(arr.size) // 3
    for (e in arr) {
        print(e)
    }
    // ADE
}
```

All the above operations are the same as for `MutableList`, but this is where the list of basic array operations ends. Arrays do not support equality, so two arrays with the same elements are not considered equal. Another problem with arrays is that their `toString` method, which is used to transform an object into a `String`, does not print elements. It only prints the array type and the hash of its memory reference.

```
fun main() {
    val arr1 = arrayOf("A", "B", "C")
    val arr2 = arrayOf("A", "B", "C")
    println(arr1 == arr2) // false
    println(arr1) // [Ljava.lang.String;@4f023edb
    println(arr2) // [Ljava.lang.String;@3a71f4dd
}
```

To cheer up those who like using arrays, the Kotlin standard library offers a number of extension functions that allow many kinds of array transformations.

```

fun main() {
    val arr = arrayOf("A", "B", "C")

    arr.|
}   ⚡ copyOf() for Array<T> in kotlin.collections           Array<String>
   ⚡ copyOf(newSize: Int) for Array<T> in kotlin.collections      Array<String?>
   ⚡ copyOfRange(fromIndex: Int, toIndex: Int) for Array<T> in kotlin.collections      Array<String>
   ⚡ fill(element: String, fromIndex: Int = ..., toIndex: Int = ...) for Array<T> in kotlin.collections      Unit
   ⚡ plus(element: String) for Array<T> in kotlin.collections           Array<String>
   ⚡ plus(elements: Array<out String>) for Array<T> in kotlin.collections      Array<String>
   ⚡ plus(elements: Collection<String>) for Array<T> in kotlin.collections      Array<String>
   ⚡ plusElement(element: String) for Array<T> in kotlin.collections           Array<String>
   ⚡ reverse() for Array<T> in kotlin.collections           Unit
   ⚡ reverse(fromIndex: Int, toIndex: Int) for Array<T> in kotlin.collections      Unit
   ⚡ reversedArray() for Array<T> in kotlin.collections           Array<String>
   ⚡ reversedArray() for Array<out T> in kotlin.collections      List<String>
   ⚡ shuffle() for Array<T> in kotlin.collections           Unit
   ⚡ shuffle(random: Random) for Array<T> in kotlin.collections      Unit
   ⚡ sliceArray(indices: IntRange) for Array<T> in kotlin.collections      Array<String>
   ⚡ slice(indices: Iterable<Int>) for Array<out T> in kotlin.collections      List<String>
   ⚡ slice(indices: IntRange) for Array<out T> in kotlin.collections      List<String>
   ⚡ sliceArray(indices: Collection<Int>) for Array<T> in kotlin.collections      Array<String>
   ⚡ sortedArray() for Array<T> in kotlin.collections           Array<String>
   ⚡ sort() for Array<out T> in kotlin.collections           Unit
   ⚡ sort() for Array<out T> in kotlin.collections           Unit
   ⚡ sort(fromIndex: Int = ..., toIndex: Int = ...) for Array<out T> in kotlin.collections      Unit
   ⚡ sort(fromIndex: Int = ..., toIndex: Int = ...) for Array<out T> in kotlin.collections      Unit
   ⚡ sorted() for Array<out T> in kotlin.collections           List<String>
   ⚡ sortedArrayDescending() for Array<T> in kotlin.collections      Array<String>
   ⚡ flatMap<T> (transform: (String) -> TCollection<T>) for Array<out T> in kotlin.collections      List<String>
   ⚡ flatMap<T> (transform: (String) -> TCollection<T>) for Array<out T> in kotlin.collections      List<String>
Press ⌘ to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

```

Notice that there is a `plus` method that allows a new element to be added to an array. Just like the `plus` method on a list, it does not modify the array but creates a new one with a larger size.

```

// JVM implementation
operator fun <T> Array<T>.plus(element: T): Array<T> {
    val index = size
    val result = java.util.Arrays.copyOf(this, index + 1)
    result[index] = element
    return result
}

fun main() {
    val arr = arrayOf("A", "B", "C")
    println(arr.size) // 3
    val arr2 = arr + "D"
    println(arr.size) // 3
}

```

```
    println(arr2.size) // 4
}
```

You can transform an array to a list or a set using the `toList` and `toSet` methods. To transfer the other way around, use `toTypedArray`.

```
fun main() {
    val arr1: Array<String> = arrayOf("A", "B", "C")
    val list: List<String> = arr1.toList()
    val arr2: Array<String> = list.toTypedArray()
    val set: Set<String> = arr2.toSet()
    val arr3: Array<String> = set.toTypedArray()
}
```

Arrays of primitives

Some kinds of Kotlin value types, like `Int` or `Char`, can be represented in a more basic way than a regular object. This form is known as a primitive and is a Kotlin optimization that does not affect the usage of values; however, it makes primitive values take less memory and their use more efficient. The problem is that primitives cannot be kept in regular collections, but we can store them in special arrays. For each value that has a primitive form, there is a dedicated array type. These are:

- `IntArray`, which represents an array of primitive `Int` values.
- `LongArray`, which represents an array of primitive `Long` values.
- `DoubleArray`, which represents an array of primitive `Double` values.
- `FloatArray`, which represents an array of primitive `Float` values.
- `CharArray`, which represents an array of primitive `Char` values.
- `BooleanArray`, which represents an array of primitive `Boolean` values.

- `ShortArray`, which represents an array of primitive `Short` values.
- `ByteArray`, which represents an array of primitive `Byte` values.

Each of these arrays can be created in two ways:

- Using the `xxxOf` function and initial elements as arguments, where `xxx` is the decapitalized name of the array. For example, to create `DoubleArray`, you can use the `doubleArrayOf` function with arguments of type `Double`.
- By transforming another kind of collection into an array of primitives using the `toXXX` method, where `xxx` is the name of the array. For instance, you can transform `List<Boolean>` into `BooleanArray` using the `toBooleanArray` method.

```
fun main() {  
    val doubles: DoubleArray = doubleArrayOf(2.71, 3.14, 9.8)  
    val chars: CharArray = charArrayOf('X', 'Y', 'Z')  
  
    val accepts: List<Boolean> = listOf(true, false, true)  
    val acceptsArr: BooleanArray = accepts.toBooleanArray()  
  
    val ints: Set<Int> = setOf(2, 4, 8, 10)  
    val intsArr: IntArray = ints.toIntArray()  
}
```

Arrays of primitives are not used often in most real-life projects. They are generally treated as low-level performance or memory use optimizations⁷³.

⁷³See Effective Kotlin, Item 55: Consider Arrays with primitives for performance-critical processing.

Vararg parameters and array functions

As mentioned in the chapter Functions, we can use the `vararg` modifier for a parameter to make it accept any number of arguments. This modifier turns a parameter into an array. Consider the `markdownList` function from the example below. Its `lines` parameter has `String` type specified, but since it has the modifier `vararg`, the actual type of `lines` is `Array<String>`. This is why we can iterate over it using a `for` loop.

```
fun markdownList(vararg lines: String): String {
    // the type of lines is Array<String>
    var str = ""
    for ((i, line) in lines.withIndex()) {
        str += " * $line"
        if (i != lines.size) {
            str += "\n"
        }
    }
    return str
}

fun main() {
    print(markdownList("A", "B", "C"))
    // * A
    // * B
    // * C
}
```

The basic functions used to create collections, like `listof` or `setof`, can have any number of arguments because they use the `vararg` modifier.

```
fun <T> listOf(vararg elements: T): List<T> =  
    if (elements.size > 0) elements.asList() else emptyList()  
  
fun <T> setOf(vararg elements: T): Set<T> =  
    if (elements.size > 0) elements.toSet() else emptySet()
```

You can also spread an array into vararg arguments using the `*` symbol.

```
fun main() {  
    val arr = arrayOf("B", "C")  
    print(markdownList("A", *arr, "D"))  
    // * A  
    // * B  
    // * C  
    // * D  
}
```

Summary

In this chapter we've seen the most important kinds of Kotlin collections and their typical use-cases:

- `List` represents an ordered collection of elements. It is the most basic way to keep a collection of elements.
- `Set` represents a collection of unique elements. We use it when we want to make sure that elements in our collection are unique, or when we often need to look for a certain element.
- `Map` is a set of key-value pairs. We use it to keep associations from keys to values.

Arrays are rarely used directly in Kotlin as we prefer to use other kinds of collections.

Operator overloading

In Kotlin, we can add an element to a list using the `+` operator. In the same way, we can add two strings together. We can check if a collection contains an element using the `in` operator. We can also add, subtract or multiply elements of type `BigDecimal`, which is a JVM class that is used to represent possibly big numbers with unlimited precision.

```
import java.math.BigDecimal

fun main() {
    val list: List<String> = listOf("A", "B")
    val newList: List<String> = list + "C"
    println(newList) // [A, B, C]

    val str1: String = "AB"
    val str2: String = "CD"
    val str3: String = str1 + str2
    println(str3) // ABCD

    println("A" in list) // true
    println("C" in list) // false

    val money1: BigDecimal = BigDecimal("12.50")
    val money2: BigDecimal = BigDecimal("3.50")
    val money3: BigDecimal = money1 * money2
    println(money3) // 43.7500
}
```

Using operators between objects is possible thanks to the Kotlin feature called *operator overloading*, which allows special kinds of methods to be defined that can be used as operators. Let's see this in a custom class example.

An example of operator overloading

Let's say that you need to represent complex numbers in your application. These are special kinds of numbers in mathematics that are represented by two parts: real and imaginary. Complex numbers are useful for a variety of kinds of calculations in physics and engineering.

```
data class Complex(val real: Double, val imaginary: Double)
```

In mathematics, there is a range of operations that we can do on complex numbers. For instance, you can add two complex numbers or subtract a complex number from another complex number. This is done using the + and - operators. Therefore, it is reasonable that we should support these operators for our `Complex` class. To support the + operator, we need to define a method that has an `operator` modifier that is called `plus` and a single parameter. To support the - operator, we need to define a method that has an `operator` modifier called `minus` and a single parameter.

```
data class Complex(val real: Double, val imaginary: Double) {

    operator fun plus(another: Complex) = Complex(
        real + another.real,
        imaginary + another.imaginary
    )

    operator fun minus(another: Complex) = Complex(
        real = real - another.real,
        imaginary = imaginary - another.imaginary
    )
}

// example usage
fun main() {
    val c1 = Complex(1.0, 2.0)
    val c2 = Complex(2.0, 3.0)
```

```
    println(c1 + c2) // Complex(real=3.0, imaginary=5.0)
    println(c2 - c1) // Complex(real=1.0, imaginary=1.0)
}
```

Using the `+` and `-` operators is equivalent to calling the `plus` and `minus` functions. These two can be used interchangeably.

```
c1 + c2 // under the hood is c1.plus(c2)
c1 - c2 // under the hood is c1.minus(c2)
```

Kotlin defines a concrete set of operators, for each of which there is a specific name and a number of supported arguments. Additionally, all operators need to be a method (so, either a member function or an extension function), and these methods need the `operator` modifier.

Well-used operators can help us improve our code readability as much as poorly used operators can harm it⁷⁴. Let's discuss all the Kotlin operators.

Arithmetic operators

Let's start with arithmetic operators, like `plus` or `times`. These are easiest for the Kotlin compiler because it just needs to transform the left column to the right.

Expression	Translates to
<code>a + b</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b)</code>
<code>a..b</code>	<code>a.rangeTo(b)</code>

⁷⁴You can find more about this in Effective Kotlin, Item 12: An operator's meaning should be consistent with its function name and Item 13: Use operators to increase readability.

Notice that `%` translates to `rem`, which is a short form of “remainder”. This operator returns the remainder left over when one operand is divided by a second operand, so it is similar to the modulo operation⁷⁵.

```
fun main() {  
    println(13 % 4) // 1  
    println(7 % 4) // 3  
    println(1 % 4) // 1  
    println(0 % 4) // 0  
    println(-1 % 4) // -1  
    println(-5 % 4) // -1  
    println(-7 % 4) // -3  
}
```

Another interesting operator is `rangeTo`, thanks to which you can create a range by using two dots between two values. When we use `rangeTo` between two numbers of type `Int`, the result is `IntRange`. To create a `ClosedRange`, you can use `..` between any two numbers that are comparable.

```
fun main() {  
    val intRange: IntRange = 1..10  
    val comparableRange: ClosedRange<String> = "A".."Z"  
}
```

The `rangeUntil` operator

Kotlin 1.7.20 introduced experimental support for a new operator called `rangeUntil`, which is basically a replacement for

⁷⁵This operator was previously called `mod`, which comes from “modulo”, but this name is now deprecated. In mathematics, both the remainder and the modulo operations act the same for positive numbers, but the difference lies in negative numbers. The result of $-5 \text{ remainder } 4$ is -1 , because $-5 = 4 * (-1) + (-1)$. The result of $-5 \text{ modulo } 4$ is 3 , because $-5 = 4 * (-2) + 3$. Kotlin’s `%` operator implements the behavior of remainder, which is why its name needed to be changed from `mod` to `rem`.

the `until` function. It is implemented using the `rangeUntil` function and can be used with the `.. operator.`

```
fun main() {
    for (a in 1..<5) {
        println(a)
    }
}
// 1
// 2
// 3
// 4
```

The `in` operator

One of my favorite operators is `in`. The expression `a in b` translates to `b.contains(a)`. There is also `!in`, which translates to negation.

Expression	Translates to
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

There are a few ways to use this operator. Firstly, for collections, instead of checking if a list contains an element, you can check if the element is in the list.

```
fun main() {
    val letters = setOf("A", "B", "C")
    println("A" in letters) // true
    println("D" in letters) // false
    println(letters.contains("A")) // true
    println(letters.contains("D")) // false
}
```

Why would you do that? Primarily for readability. Would you ask “Does the fridge contain a beer?” or “Is there a beer in

the fridge”? Using the `in` operator gives us the possibility to choose.

We also often use the `in` operator together with ranges. The expression `1..10` produces an object of type `IntRange`, which has a `contains` method. This is why you can use `in` and a range to check if a number is in this range.

```
fun main() {
    println(5 in 1..10) // true
    println(11 in 1..10) // false
}
```

You can make a range from any objects that are comparable, and the result `ClosedRange` also has a `contains` method. This is why you can use a range check for any objects that are comparable, such as big numbers or objects representing time.

```
import java.math.BigDecimal
import java.time.LocalDateTime

fun main() {
    val amount = BigDecimal("42.80")
    val minPrice = BigDecimal("5.00")
    val maxPrice = BigDecimal("100.00")
    val correctPrice = amount in minPrice..maxPrice
    println(correctPrice) // true

    val now = LocalDateTime.now()
    val actionStarts = LocalDateTime.of(1410, 7, 15, 0, 0)
    val actionEnds = actionStarts.plusDays(1)
    println(now in actionStarts..actionEnds) // false
}
```

The iterator operator

You can use for-loop to iterate over any object that has an `iterator` operator method. Every object that implements an `Iterable` interface must support the `iterator` method.

```
public interface Iterable<out T> {  
    /**  
     * Returns an iterator over the elements of this object.  
     */  
    public operator fun iterator(): Iterator<T>  
}
```

You can define objects that can be iterated over, but do not implement `Iterable` interface. `Map` is a great example. It does not implement the `Iterable` interface, yet you can iterate over it using a for-loop. How so? It is thanks to the `iterator` operator, which is defined as an extension function in Kotlin stdlib.

```
// Part of Kotlin standard library  
inline operator fun <K, V>  
Map<out K, V>.iterator(): Iterator<Map.Entry<K, V>> =  
    entries.iterator()  
  
fun main() {  
    val map = mapOf('a' to "Alex", 'b' to "Bob")  
    for ((letter, name) in map) {  
        println("$letter like in $name")  
    }  
}  
// a like in Alex  
// b like in Bob
```

To better understand how a for-loop works, consider the code below.

```
fun main() {
    for (e in Tree()) {
        // body
    }
}

class Tree {
    operator fun iterator(): Iterator<String> = ...
}
```

Under the hood, a for-loop is compiled into bytecode that uses a while-loop to iterate over the object's iterator, as presented in the snippet below.

```
fun main() {
    val iterator = Tree().iterator()
    while (iterator.hasNext()) {
        val e = iterator.next()
        // body
    }
}
```

The equality and inequality operators

In Kotlin, there are two types of equality:

- Structural equality - checked with the `equals` method or the `==` operator (and its negated counterpart `!=`). `a == b` translates to `a.equals(b)` when `a` is not nullable, otherwise it translates to `a?.equals(b) ?: (b == null)`. Structural equality is generally preferred over referential equality. The `equals` method can be overridden in custom class.
- Referential equality - checked with the `===` operator (and its negated counterpart `!==`); returns `true` when both sides point to the same object. `==` and `!=` (identity checks) are not overloadable.

Since `equals` is implemented in `Any`, which is the superclass of every class, we can check the equality of any two objects.

Expression	Translates to
<code>a == b</code>	<code>a?.equals(b) ?: (b === null)</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: (b === null))</code>

Comparison operators

Some classes have natural order, which is the order that is used by default when we compare two instances of a given class. Numbers are a good example: 10 is a smaller number than 100. There is a popular Java convention that classes with natural order should implement a `Comparable` interface that requires a `compareTo` method that is used to compare two objects.

```
public interface Comparable<in T> {  
    /**  
     * Compares this object with the specified object for  
     * order. Returns zero if this object is equal to the  
     * specified [other] object, a negative number if it's  
     * less than [other], or a positive number if it's  
     * greater than [other].  
     */  
    public operator fun compareTo(other: T): Int  
}
```

As a result, there is a convention that we should compare two objects using the `compareTo` method. However, using the `compareTo` method directly is not very intuitive. Let's say that you see `a.compareTo(b) > 0` in code. What does it mean? Kotlin simplifies this by making `compareTo` an operator that can be replaced with intuitive mathematical comparison operators: `>`, `<`, `>=`, and `<=`.

Expression	Translates to
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

I often use comparison operators to compare amounts kept in objects of type `BigDecimal` OR `BigInteger`.

```
import java.math.BigDecimal

fun main() {
    val amount1 = BigDecimal("42.80")
    val amount2 = BigDecimal("5.00")
    println(amount1 > amount2) // true
    println(amount1 >= amount2) // true
    println(amount1 < amount2) // false
    println(amount1 <= amount2) // false
    println(amount1 > amount1) // false
    println(amount1 >= amount1) // true
    println(amount1 < amount2) // false
    println(amount1 <= amount2) // false
}
```

I also like to compare time references the same way.

```
import java.time.LocalDateTime

fun main() {
    val now = LocalDateTime.now()
    val actionStarts = LocalDateTime.of(2010, 10, 20, 0, 0)
    val actionEnds = actionStarts.plusDays(1)
    println(now > actionStarts) // true
    println(now <= actionStarts) // false
    println(now < actionEnds) // false
    println(now >= actionEnds) // true
}
```

The indexed access operator

In programming, there are two popular conventions for getting or setting elements in collections. The first uses box brackets, while the second uses the `get` and `set` methods. In Java, we use the first convention for arrays and the second one for other kinds of collections. In Kotlin, both conventions can be used interchangeably because the `get` and `set` methods are operators that can be used with box brackets.

Expression	Translates to
<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, j]</code>	<code>a.get(i, j)</code>
<code>a[i_1, ..., i_n]</code>	<code>a.get(i_1, ..., i_n)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, j] = b</code>	<code>a.set(i, j, b)</code>
<code>a[i_1, ..., i_n] = b</code>	<code>a.set(i_1, ..., i_n, b)</code>

```
fun main() {
    val mutableList = mutableListOf("A", "B", "C")
    println(mutableList[1]) // B
    mutableList[2] = "D"
    println(mutableList) // [A, B, D]

    val animalFood = mutableMapOf(
        "Dog" to "Meat",
        "Goat" to "Grass"
    )
    println(animalFood["Dog"]) // Meat
    animalFood["Cat"] = "Meat"
    println(animalFood["Cat"]) // Meat
}
```

Square brackets are translated to `get` and `set` calls with appropriate numbers of arguments. Variants of `get` and `set` functions with more arguments might be used by data processing

libraries. For instance, you could have an object that represents a table and use box brackets with two arguments: x and y coordinates.

Augmented assignments

When we set a new value for a variable, this new value is often based on its previous value. For instance, we might want to add a value to the previous one. For this, augmented assignments were introduced⁷⁶. For example, $a += b$ is a shorter notation of $a = a + b$. There are similar notations for other arithmetic operations.

Expression	Translates to
$a += b$	$a = a + b$
$a -= b$	$a = a - b$
$a *= b$	$a = a * b$
$a /= b$	$a = a / b$
$a %= b$	$a = a \% b$

Notice that augmented assignments can be used for all types that support the appropriate arithmetic operation, including lists or strings. Such augmented assignments need a variable to be read-write, namely `var`, and the result of the mathematical operation must have a proper type (to translate $a += b$ to $a = a + b$, the variable a needs to be `var`, and $a + b$ needs to be a subtype of type a).

⁷⁶I am not sure which language introduced augmented assignments first, but they are even supported by languages as old as C.

```

fun main() {
    var str = "ABC"
    str += "D" // translates to str = str + "D"
    println(str) // ABCD

    var l = listOf("A", "B", "C")
    l += "D" // translates to l = l + "D"
    println(l) // [A, B, C, D]
}

```

Augmented assignments can be used in another way: to modify a mutable object. For instance, we can use `+=` to add an element to a mutable list. In such a case, `a += b` translates to `a.plusAssign(b)`.

Expression	Translates to
<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.remAssign(b)</code>

```

fun main() {
    val names = mutableListOf("Jake", "Ben")
    names += "Jon"
    names -= "Ben"
    println(names) // [Jake, Jon]

    val tools = mutableMapOf(
        "Grass" to "Lawnmower",
        "Nail" to "Hammer"
    )
    tools += "Screw" to "Screwdriver"
    tools -= "Grass"
    println(tools) // {Nail=Hammer, Screw=Screwdriver}
}

```

If both kinds of augmented assignment can be applied, Kotlin chooses to modify a mutable object by default.

Unary prefix operators

A plus, minus, or negation in front of a single value is also an operator. Operators that are used with only a single value are called **unary operators**⁷⁷. Kotlin supports operator overloading for the following unary operators:

Expression	Translates to
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

Here is an example of overloading the `unaryMinus` operator.

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

fun main() {
    val point = Point(10, 20)
    println(-point) // prints "Point(x=-10, y=-20)"
}
```

The `unaryPlus` operator is often used as part of Kotlin DSLs, which are described in detail in the next book of this series, *Functional Kotlin*.

⁷⁷Unary operators are used with only a single value (operand). Operators used with two values are known as binary operators; however, since most operators are binary, this type is often treated as the default. Operators used with three values are known as ternary operators. Since there is only one ternary operator in mainstream programming languages, namely the **conditional operator**, it is often referred as **the ternary operator**.

Increment and decrement

As part of many algorithms used in older languages, we often needed to add or subtract the value 1 from a variable, which is why increment and decrement were invented. The `++` operator is used to add 1 to a variable; so, if `a` is an integer, then `a++` translates to `a = a + 1`. The `--` operator is used to subtract 1 from a variable; so, if `a` is an integer, then `a--` translates to `a = a - 1`.

Both increment and decrement can be used before or after a variable, and this determines the value returned by this operation.

- If you use `++ before` a variable, it is called **pre-increment**; it increments the variable and then returns the result of this operation.
- If you use `++ after` a variable, it is called **post-increment**; it increments the variable but then returns the value before the operation.
- If you use `-- before` a variable, it is called **pre-decrement**; it decrements the variable and then returns the result of this operation.
- If you use `-- after` a variable, it is called **post-decrement**; it decrements the variable but then returns the value before the operation.

```
fun main() {  
    var i = 10  
    println(i++) // 10  
    println(i) // 11  
    println(++i) // 12  
    println(i) // 12  
  
    i = 10  
    println(i--) // 10  
    println(i) // 9  
    println(--i) // 8
```

```
    println(i) // 8
}
```

Based on the `inc` and `dec` methods, Kotlin supports increment and decrement overloading, which should increment or decrement a custom object. I have never seen this capability used in practice, so I think it is enough to know that it exists.

Expression	Translates to (simplified)
<code>++a</code>	<code>a.inc(); a</code>
<code>a++</code>	<code>val tmp = a; a.inc(); tmp</code>
<code>--a</code>	<code>a.dec(); a</code>
<code>a--</code>	<code>val tmp = a; a.dec(); tmp</code>

The invoke operator

Objects with the `invoke` operator can be called like functions, so with parentheses straight after the variable representing this object. Calling an object translates to the `invoke` method call with the same arguments.

Expression	Translates to
<code>a()</code>	<code>a.invoke()</code>
<code>a(i)</code>	<code>a.invoke(i)</code>
<code>a(i, j)</code>	<code>a.invoke(i, j)</code>
<code>a(i_1, ..., i_n)</code>	<code>a.invoke(i_1, ..., i_n)</code>

The `invoke` operator is used for objects that represent functions, such as lambda expressions⁷⁸ or UseCases objects from Clean Architecture.

⁷⁸There will be more about lambda expressions in the next book of the series, *Functional Kotlin*.

```

class CheerUseCase {
    operator fun invoke(who: String) {
        println("Hello, $who")
    }
}

fun main() {
    val hello = {
        println("Hello")
    }
    hello() // Hello

    val cheerUseCase = CheerUseCase()
    cheerUseCase("Reader") // Hello, Reader
}

```

Precedence

What is the result of the expression `1 + 2 * 3`? The answer is 7, not 9, because in mathematics we multiply before adding. We say that multiplication has higher precedence than addition.

Precedence is also extremely important in programming because when the compiler evaluates an expression such as `1 + 2 == 3`, it needs to know if it should first add 1 to 2, or compare 2 and 3. The following table compares the precedence of all the operators, including those that can be overloaded and those that cannot.

Precedence	Title	Symbols
Highest	Postfix	<code>++, -, . (regular call), ?. (safe call)</code>
	Prefix	<code>-, +, ++, -, !</code>
	Type casting	<code>as, as?</code>
	Multiplicative	<code>*, /, %</code>
	Additive	<code>+, -</code>
	Range	<code>..</code>

Precedence	Title	Symbols
	Infix function	simpleIdentifier
	Elvis	?:
	Named checks	in, !in, is, !is
	Comparison	<, >, <=, >=
	Equality	==, !=, ===, !==
	Conjunction	&&
	Disjunction	\
	Spread operator	*
Lowest	Assignment	=, +=, -=, *=, /=, %= %+=

On the basis of this table, can you predict what the following code will print?

```
fun main() {
    println(-1.plus(1))
}
```

This is a popular Kotlin puzzle. The answer is `-2`, not `0`, because a single minus in front of a function is an operator whose precedence is lower than an explicit `plus` method call. So, we first call the method and then call `unaryMinus` on the result, therefore we change from `2` to `-2`. To use `-1` literally, wrap it with parentheses.

```
fun main() {
    println((-1).plus(1)) // 0
}
```

Summary

We use a lot of operators in Kotlin, many of which can be overloaded. This can be used to improve our code's readability. From the cognitive standpoint, using an intuitive operator can be a huge improvement over using methods everywhere. Therefore, it's good to know what options are available and to be open to using operators defined by Kotlin stdlib, but it's also good to be able to define our own operators.

The beauty of Kotlin's type system

The Kotlin type system is amazingly designed. Many features that look like special cases are just a natural consequence of how the type system is designed. For instance, thanks to the type system, in the example below the type of `surname` is `String`, the type of `age` is `Int`, and we can use `return` and `throw` on the right side of the Elvis operator.

```
fun processPerson(person: Person?) {
    val name = person?.name ?: "unknown"

    val surname = person?.surname ?: return

    val age = person?.age
        ?: throw Error("Person must have age")

    // ...
}
```

The typing system also gives us very convenient nullability support, smart type inference, and much more. In this chapter, we will reveal a lot of Kotlin magic. I always love talking about this in my workshops because I see the stunning beauty of how Kotlin's type system is so well designed that all these pieces fit perfectly together and give us a great programming experience. I find this topic fascinating, but I will also try to add some useful hints that show where this knowledge can be useful in practice. I hope you will enjoy discovering it as much as I did.

What is a type?

Before we start talking about the type system, we should first explain what a type is. Do you know the answer? Think about it for a moment.

Types are commonly confused with classes, but these two terms represent totally different concepts. Take a look at the example below. You can see `User` used four times. Can you tell me which usages are classes, which are types, and which are something else?

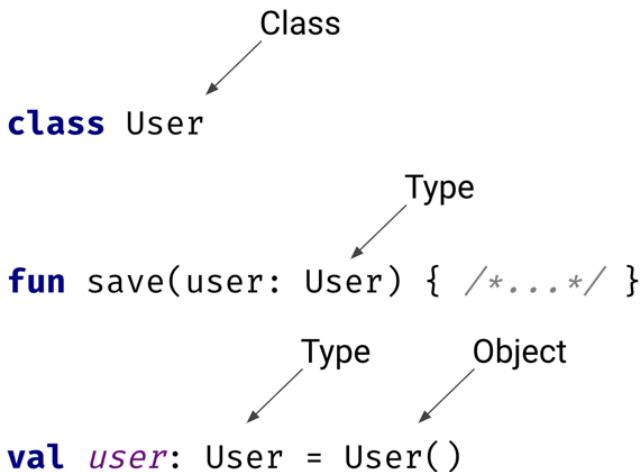
```
class User

fun save(user: User) { /*...*/ }

val user: User = User()
```

After the `class` keyword, you define a class name. A class is a template for objects that defines a set of properties and methods. When we call a constructor, we create an object. Types are used here to specify what kind of objects we expect to have in the variables⁷⁹.

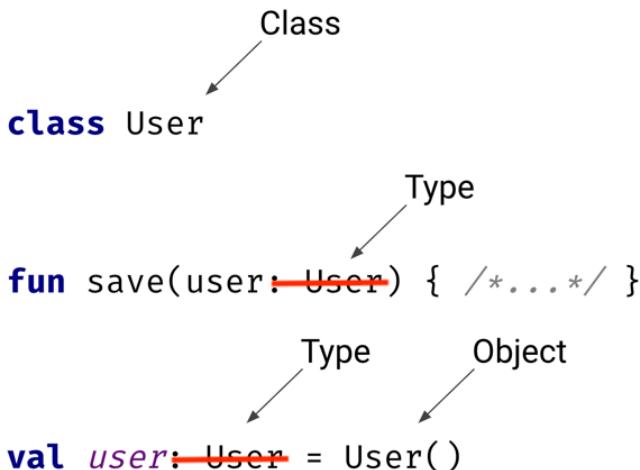
⁷⁹Parameters are also variables.



Why do we have types?

Let's do a thought experiment for a moment. Kotlin is a statically typed language, so all variables and functions must be typed. If we do not specify their types explicitly, they will be inferred. But let's take a step back and imagine that you are a language designer who is deciding what Kotlin should look like. It is possible to drop all these requirements and eliminate all types completely. The compiler does not really need them⁸⁰. It has classes that define how objects should be created, and it has objects that are used during execution. What do we lose if we get rid of types? Mostly safety and developers' convenience.

⁸⁰Except when figuring out which function to choose in the case of overloading.



It is worth mentioning that many languages do support classes and objects but not types. Among them, there is JavaScript⁸¹ and (not long ago) Python - two of the most popular languages in the world⁸². However, types do offer us value, which is why in the JavaScript community more and more people use TypeScript (which is basically JavaScript plus types), and Python has introduced support for types.

So why do we have types? They are mainly for us, developers. A type tells us what methods or properties we can use on an object. A type tells us what kind of value can be used as an argument. Types prevent the use of incorrect objects, methods, or properties. They give us safety, and suggestions

⁸¹Formally, JavaScript supports weak typing, but in this chapter we discuss static typing, which is not supported by JavaScript.

⁸²It all depends on what we measure, but Python, Java, and JavaScript take the first three positions in most rankings. In some, they are beaten by C, which is widely used for very low-level development, like developing processors for cars or refrigerators.

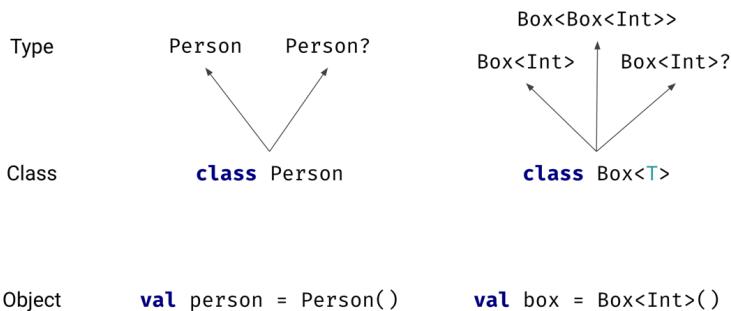
are provided by the IDE. The compiler also benefits from types as they are used to better optimize our code or to decide which function should be chosen when its name is overloaded. Still, it is developers who are the most important beneficent of types.

So what is a type? **It can be considered as a set of things we can do with an object.** Typically, it is a set of methods and properties.

The relation between classes and types

We say that classes generate types. Think of the class `User`. It generates two types. Can you name them both? One is `User`, but the second is not `Any` (`Any` is already in the type hierarchy). The second new type generated by the class `User` is `User?`. Yes, the nullable variant is a separate type.

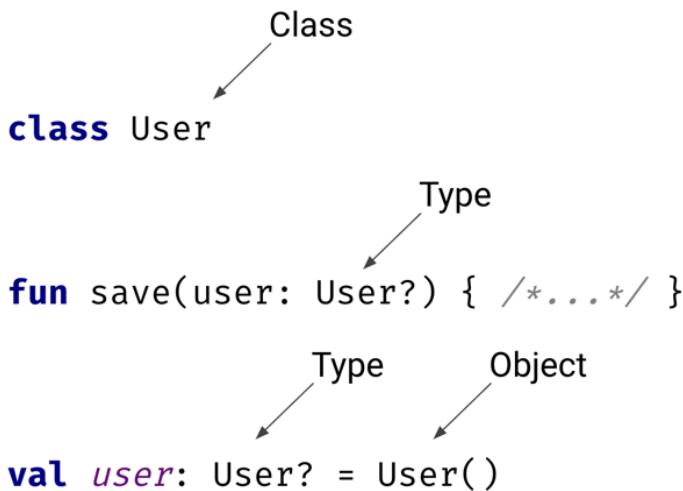
There are classes that generate many more types: generic classes. The `Box<T>` class theoretically generates an infinite number of types.



Class vs type in practice

This discussion might sound very theoretical, but it already has some practical implications. Note that classes cannot be nullable, but types can. Consider the initial example, where I

asked you to point out where `User` is a type. Only in positions that represent types can you use `User?` instead of `User`.



Member functions are defined on classes, so their receiver cannot be nullable or have type arguments⁸³. Extension functions are defined on types, so they can be nullable or defined on a concrete generic type. Consider the `sum` function,, which is an extension of `Iterable<Int>`, or the `isNullOrEmpty` function, which is an extension of `String?`.

⁸³Type arguments and type parameters will be better explained in the chapter *Generics*.

```
fun Iterable<Int>.sum(): Int {
    var sum: Int = 0
    for (element in this) {
        sum += element
    }
    return sum
}

@OptIn(ExperimentalContracts::class)
inline fun CharSequence?.isNullOrBlank(): Boolean {
    contract {
        returns(false) implies (this@isNullOrBlank != null)
    }

    return this == null || this.isBlank()
}
```

The relationship between types

Let's say that we have a class `Dog` and its superclass `Animal`.

```
open class Animal
class Dog : Animal()
```

Wherever an `Animal` type is expected, you can use a `Dog`, but not the other way around.

```
fun petAnimal(animal: Animal) {}
fun petDog(dog: Dog) {}

fun main() {
    val dog: Dog = Dog()
    val dogAnimal: Animal = dog // works
    petAnimal(dog) // works
    val animal: Animal = Animal()
    val animalDog: Dog = animal // compilation error
    petDog(animal) // compilation error
}
```

Why? Because there is a concrete relationship between these types: `Dog` is a subtype of `Animal`. By rule, when A is a subtype of B, we can use A where B is expected. We might also say that `Animal` is a supertype of `Dog`, and a subtype can be used where a supertype is expected.



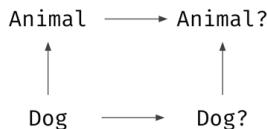
There is also a relationship between nullable and non-nullable types. A non-nullable can be used wherever a nullable is expected.

```

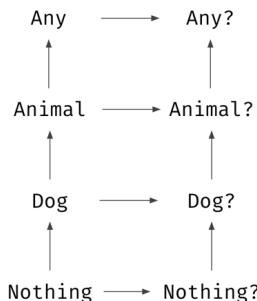
fun petDogIfPresent(dog: Dog?) {}
fun petDog(dog: Dog) {}

fun main() {
    val dog: Dog = Dog()
    val dogNullable: Dog? = dog
    petDogIfPresent(dog) // works
    petDogIfPresent(dogNullable) // works
    petDog(dog) // works
    petDog(dogNullable) // compilation error
}
    
```

This is because the non-nullable variant of each type is a subtype of the nullable variant.



The superclass of all the classes in Kotlin is `Any`, which is similar to `object` in Java. The supertype of all the types is not `Any`, it is `Any?`. `Any` is a supertype of all non-nullable types. We also have something that is not present in Java and most other mainstream languages: the subtype of all the types, which is called `Nothing`. We will talk about it soon.



`Any` is only a supertype of non-nullable types. So, wherever `Any` is expected, nullable types will not be accepted. This fact is also used to set a type parameter's upper boundary to accept only non-nullable types⁸⁴.

```
fun <T : Any> String.parseJson(): T = ...
```

`Unit` does not have any special place in the type hierarchy. It is just an object declaration that is used when a function does not specify a result type.

```
object Unit {
    override fun toString() = "kotlin.Unit"
}
```

Let's talk about a concept that has a very special place in the typing hierarchy: let's talk about `Nothing`.

⁸⁴I will explain type parameters' upper boundaries in the chapter `Generics`.

The subtype of all the types: Nothing

`Nothing` is a subtype of all the types in Kotlin. If we had an instance of this type, it could be used instead of everything else (like a Joker in the card game Rummy). It's no wonder that such an instance does not exist. `Nothing` is an empty type (also known as a bottom type, zero type, uninhabited type, or never type), which means it has no values. It is literally impossible to make an instance of type `Nothing`, but this type is still really useful. I will tell you more: some functions declare `Nothing` as their result type. You've likely used such functions many times already. What functions are those? They declare `Nothing` as a result type, but they cannot return it because this type has no instances. But what can these functions do? Three things: they either need to run forever, end the program, or throw an exception. In all cases, they never return, so the `Nothing` type is not only valid but also really useful.

```
fun runForever(): Nothing {
    while (true) {
        // no-op
    }
}

fun endProgram(): Nothing {
    exitProcess(0)
}

fun fail(): Nothing {
    throw Error("Some error")
}
```

I have never found a good use case for a function that runs forever, and ending a program is not very common, but we often use functions that throw exceptions. Who hasn't ever used `TODO()`? This function throws a `NotImplementedError` exception. There is also the `error` function from the standard library, which throws an `IllegalStateException`.

```
inline fun TODO(): Nothing = throw NotImplementedError()

inline fun error(message: Any): Nothing =
    throw IllegalStateException(message.toString())
```

TODO is used as a placeholder in a place where we plan to implement some code.

```
fun fib(n: Int): Int = TODO()
```

error is used to signal an illegal situation:

```
fun get(): T = when {
    left != null -> left
    right != null -> right
    else -> error("Must have either left or right")
}
```

This result type is significant. Let's say that you have an if-condition that returns either Int or Nothing. What should the inferred type be? The closest supertype of both Int and Nothing is Int. This is why the inferred type will be Int.

```
// the inferred type of answer is Int
val answer = if (timeHasPassed) 42 else TODO()
```

The same rule applies when we use the Elvis operator, a when-expression, etc. In the example below, the type of both name and fullName is String because both fail and error declare Nothing as their result type. This is a huge convenience.

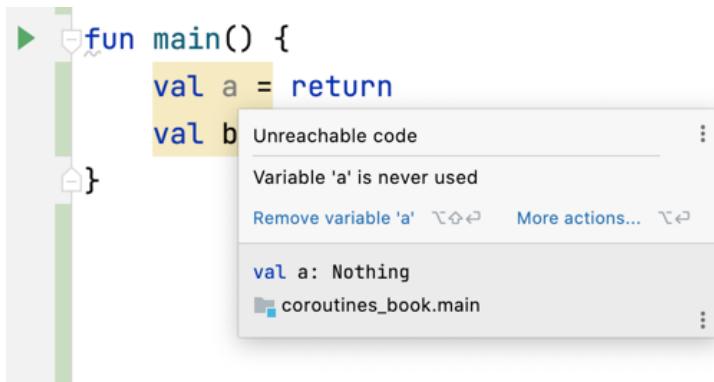
```
fun processPerson(person: Person?) {  
    // the inferred type of name is String  
    val name = person?.name ?: fail()  
    // the inferred type of fullName is String  
    val fullName = when {  
        !person.middleName.isNullOrEmpty() ->  
            "$name ${person.middleName} ${person.surname}"  
        !person.surname.isNullOrEmpty() ->  
            "$name ${person.surname}"  
        else ->  
            error("Person must have a surname")  
    }  
    // ...  
}
```

The result type from return and throw

I will start this subchapter with something strange: did you know that you can place `return` or `throw` on the right side of a variable assignment?

```
fun main() {  
    val a = return  
    val b = throw Error()  
}
```

This doesn't make any sense as both `return` and `throw` end the function, so we will never assign anything to such variables (like `a` and `b` in the example above). This assignment is an unreachable piece of code. In Kotlin, it just causes a warning.



The code above is correct from the language point of view because both `return` and `throw` are expressions, which means they declare a result type. This type is `Nothing`.

```
fun main() {
    val a: Nothing = return
    val b: Nothing = throw Error()
}
```

This explains why we can place `return` or `throw` on the right side of the Elvis operator or in a `when`-expression.

```
fun processPerson(person: Person?) {
    val name = person?.name ?: return
    val fullName = when {
        !person.middleName.isNullOrEmpty() ->
            "$name ${person.middleName} ${person.surname}"
        !person.surname.isNullOrEmpty() ->
            "$name ${person.surname}"
        else -> return
    }
    // ...
}
```

```
fun processPerson(person: Person?) {  
    val name = person?.name ?: throw Error("Name is required")  
    val fullName = when {  
        !person.middleName.isNullOrEmpty() ->  
            "$name ${person.middleName} ${person.surname}"  
        !person.surname.isNullOrEmpty() ->  
            "$name ${person.surname}"  
        else -> throw Error("Surname is required")  
    }  
    // ...  
}
```

Both `return` and `throw` declare `Nothing` as their result type. As a consequence, Kotlin will infer `String` as the type of both `name` and `fullName` because `String` is the closest supertype of both `String` and `Nothing`.

So, now you can say that you know `Nothing`. Just like John Snow.



When is some code not reachable?

When an element declares `Nothing` as a return type, it means that everything after its call is not reachable. This is reasonable: there are no instances of `Nothing`, so it cannot be returned. This means a statement that declares `Nothing` as its result type will never complete in a normal way, so the next statements are not reachable. This is why everything after either `fail` or `throw` will be unreachable.

```
fun test1() {
    print("Before")
    fail()
    print("After")
}

fun test2() {
    print("Before")
    throw Error()
    print("After")
}
```

↑
Unreachable code

It's the same with `return`, `TODO`, `error`, etc. If a non-optional expression declares `Nothing` as its result type, everything after that is unreachable. This is a simple rule, but it's useful for the compiler. It's also useful for us since it gives us more possibilities. Thanks to this rule, we can use `TODO()` in a function instead of returning a value. Anything that declares `Nothing` as a result type ends the function (or runs forever), so this function will not end without returning or throwing first.

```
fun fizzBuzz(): String {  
    TODO()  
}
```

I would like to end this topic with a more advanced example that comes from the Kotlin Coroutines library. There is a `MutableStateFlow` class, which represents a mutable value whose state changes can be observed using the `collect` method. The thing is that `collect` suspends the current coroutine until whatever it observes is closed, but a `StateFlow` cannot be closed. This is why this `collect` function declares `Nothing` as its result type.

```
public interface SharedFlow<out T> : Flow<T> {  
    public val replayCache: List<T>  
    override suspend fun collect(  
        collector: FlowCollector<T>  
    ): Nothing  
}
```

That is very useful for developers who are not aware of how `collect` works. Thanks to the result type, IntelliJ informs them that the code they place after `collect` is unreachable.

```
suspend fun main(): Unit = coroutineScope { this: CoroutineScope  
    val state = MutableStateFlow( value: 0 )  
    state.collect { it: Int  
        println(it)  
    }  
  
    state.value = 10  
    state.value = 20 Unreachable code  
}
```

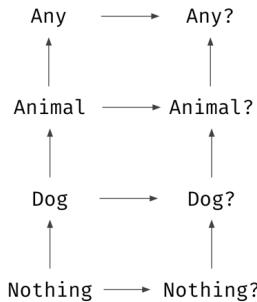
`SharedFlow` cannot be closed, so its `collect` function will never return, therefore it declares `Nothing` as its result type.

The type of null

Let's see another peculiar thing. Did you know that you can assign `null` to a variable without setting an explicit type? What's more, such a variable can be used wherever `null` is accepted.

```
fun main() {  
    val n = null  
    val i: Int? = n  
    val d: Double? = n  
    val str: String? = n  
}
```

This means that `null` has its type, which is a subtype of all nullable types. Take a look at the type hierarchy and guess what type this is.



I hope you guessed that the type of `null` is `Nothing?`. Now think about the inferred type of `a` and `b` in the example below.

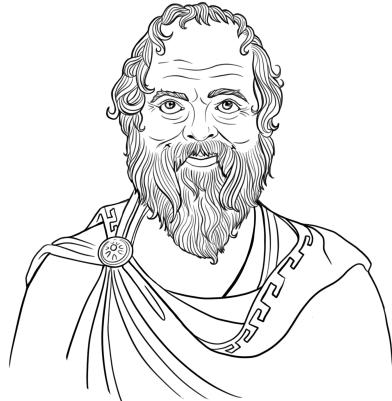
```
val a = if (predicate) "A" else null

val b = when {
    predicate2 -> "B"
    predicate3 -> "C"
    else -> null
}
```

In the if-expression, we search for the closest supertype of the types from both branches. The closest supertype of `String` and `Nothing?` is `String?`. The same is true about the when-expression: the closest supertype of `String`, `String`, and `Nothing?` is `String?`. Everything makes sense.

For the same reason, whenever we require `String?`, we can pass either `String` or `null`, whose type is `Nothing?`. This is clear when you take a look at the type hierarchy. `String` and `Nothing?` are the only non-empty subtypes of `String?`.

SOCRATES KNOWS, THAT HE KNOWS NOTHING



Summary

In this chapter, we've learned the following:

- A class is a template for creating objects. A type defines expectations and functionalities.
- Every class generates a nullable and a non-nullable type.
- A nullable type is a supertype of the non-nullable variant of this type.
- The supertype of all types is `Any?`.
- The supertype of non-nullable types is `Any`.
- The subtype of all types is `Nothing`.
- When a function declares `Nothing` as a return type, this means that it will throw an error or run infinitely.
- Both `throw` and `return` declare `Nothing` as their result type.
- The Kotlin compiler understands that when an expression declares `Nothing` as a result type, everything after that is unreachable.
- The type of `null` is `Nothing?`, which is the subtype of all nullable types.

In the next chapter, we are going to discuss generics, and we'll see how they are important for our type system.

Generics

In the early days of Java, it was designed such that all lists had the same type `List`, instead of specific lists with specific parameter types, like `List<String>` or `List<Int>`. The `List` type in Java accepts all kinds of values; when you ask for a value at a certain position, the result type is `Object` (which, in Java, is the supertype of all the types).

```
// Java
List names = new ArrayList();
names.add("Alex");
names.add("Ben");
names.add(123); // this is incorrect, but compiles
for(int i = 0; i < names.size(); i++){
    String name= (String) names.get(i); // exception at i==2
    System.out.println(name.toUpperCase());
}
```

Such lists are hard to use. We much prefer to have a list with specified types of elements. Only then can we be sure that our list contains elements of the correct type, and only then do we not need to explicitly cast these elements when we get them from a list. This was one of the main reasons Java introduced generics in version 5. In Kotlin, we do not have this problem because it was designed with generics support from the beginning, and all lists are generic, so they must specify what kinds of elements they accept. Generics are an important feature of most modern programming languages; so, in this chapter, we will discuss what they are and how we use them in Kotlin.

In Kotlin, we have three kinds of generic elements:

- generic functions,
- generic classes,
- generic interfaces.

Let's discuss them one by one.

Generic functions

Just as we can pass an argument value to a parameter, we can pass a type as a **type argument**. For this, a function needs to define one or more type parameters inside angle brackets immediately after the `fun` keyword. By convention, type parameter names are capitalized. When a function defines a type parameter, we have to specify the type arguments when calling this function. The type parameter is a placeholder for a concrete type; the type argument is the actual type that is used when a function is called. To specify type arguments explicitly, we also use angle brackets.

```
fun <T> a() {} // T is type parameter
a<Int>() // Int is used here as a type argument
a<String>() // String is used here as a type argument
```

There is a popular practice that a single type argument is called τ (from “type”); if there are multiple type arguments, they are called τ with consecutive numbers. However, this practice is not a fixed rule, and there are many other conventions for naming type parameters.

```
fun <T> a() {}
fun <T1, T2> b() {}
```

When we call a generic function, all its type arguments must be clear for the Kotlin compiler. We can either specify them explicitly, or their values can be inferred from the compiler.

```
fun <T> a() {}
fun <T1, T2> b() {}
fun <T> c(t: T) {}
fun <T1, T2> d(a: T1, b: T2) {}
fun <T> e(): T = TODO()

fun main() {
    // Type arguments specified explicitly
    a<Int>()
    a<String>()
    b<Double, Char>()
    b<Float, Long>()

    // Inferred type arguments
    c(10) // The inferred type of T is Int
    d("AAA", 10.0)
    // The inferred type of T1 is String, and of T2 is Double
    val e: Boolean = e() // The inferred type of T is Boolean
}
```

So, how are these type parameters useful? We use them primarily to specify the relationship between the arguments and the result type. For instance, we can express that the result type is the same as an argument type or that we expect two arguments of the same type.

```
import kotlin.random.Random

// The result type is the same as the argument type
fun <T> id(value: T): T = value

// The result type is the closest supertype of arguments
fun <T> randomOf(a: T, b: T): T =
    if (Random.nextBoolean()) a else b

fun main() {
    val a = id(10) // Inferred a type is Int
    val b = id("AAA") // Inferred b type is String
    val c = randomOf("A", "B") // Inferred c type is String
}
```

```
    val d = randomOf(1, 1.5) // Inferred d type is Number
}
```

Type parameters for functions are useful for the compiler since they allow it to check and correctly infer types; this makes our programs safer and makes programming more pleasurable for developers. Better parameter types and type suggestions protect us from using illegal operations and let our IDE give us better suggestions.

In the next book, *Functional Kotlin*, you will see plenty of generic function examples, especially for collection processing. Such functions are really important and useful. But, for now, let's get back to the initial motivation for introducing generics: let's talk about generic classes.

Generic classes

We can make classes generic by adding a type parameter after the class name. Such a type parameter can be used all over the class body, especially to specify properties, parameters, and result types. A type parameter is specified when we define an instance, after which it remains unchanged. Thanks to that, when you declare `ValueWithHistory<String>` and then call `setValue` in the example below, you must use an object of type `String`; when you call `currentValue`, the result object will be typed as `String`; and when you call `history`, its result is of type `List<String>`. It's the same for all other possible type arguments.

```
class ValueWithHistory<T>(
    private var value: T
) {
    private var history: List<T> = listOf(value)

    fun setValue(value: T) {
        this.value = value
        this.history += value
    }
}
```

```
fun currentValue(): T = value

fun history(): List<T> = history
}

fun main() {
    val letter = ValueWithHistory<String>("A")
    // The type of letter is ValueWithHistory<String>
    letter.setValue("B")
    // letter.setValue(123) <- this would not compile
    val l = letter.currentValue() // the type of l is String
    println(l) // B
    val h = letter.history() // the type of h is List<String>
    println(h) // [A, B]
}
```

The constructor type argument can be inferred. In the above example, we specified it explicitly, but we did not need to. This type can be inferred from the argument type.

```
val letter = ValueWithHistory("A")
// The type of letter is ValueWithHistory<String>
```

Type arguments can also be inferred from variable types. Let's say that we want to use Any as a type argument. We can specify this by specifying the type of variable letter as ValueWithHistory<Any>.

```
val letter: ValueWithHistory<Any> = ValueWithHistory("A")
// The type of letter is ValueWithHistory<Any>
```

As I mentioned in the introduction to this chapter, the most important motivation for introducing generics was to make collections with certain types of elements. Consider the `ArrayList` class from the Standard Library (stdlib). It is generic, so when we create an instance from this class we need to specify the types of elements. Thanks to that, Kotlin

protects us by expecting only values with accepted types to be added to the list, and Kotlin uses this type when we operate on the elements in the list.

```
fun main() {
    val letters = ArrayList<String>()
    letters.add("A") // the argument must be of type String
    letters.add("B") // the argument must be of type String
    // The type of letters is List<String>
    val a = letters[0] // the type of a is String
    println(a) // A
    for (l in letters) { // the type of l is String
        println(l) // first A, then B
    }
}
```

Generic classes and nullability

Notice that type arguments can be nullable, so we could create `ValueWithHistory<String?>`. In such a case, the `null` value is a perfectly valid option.

```
fun main() {
    val letter = ValueWithHistory<String?>(null)
    letter.setValue("A")
    letter.setValue(null)
    val l = letter.currentValue() // the type of l is String?
    println(l) // null
    val h = letter.history() // the type of h is List<String?>
    println(h) // [null, A, null]

    val letters = ArrayList<String?>()
    letters.add("A")
    letters.add(null)
    println(letters) // [A, null]
    val l2 = letters[1] // the type of l2 is String?
    println(l2) // null
}
```

Another thing is that when you use generic parameters inside classes or functions, you can make them nullable by adding a question mark. See the example below. The type T might or might not be nullable, depending on the type argument, but the type $T?$ is always nullable. We can assign `null` to variables of the type $T?$. Nullable generic type parameter $T?$ must be unpacked before using it as T .

```
class Box<T> {
    var value: T? = null

    fun getOrThrow(): T = value!!
}
```

The opposite can also be expressed. Since a generic type parameter might represent a nullable type, we might specify a definitely non-nullable variant of this type by adding `& Any` after the type parameter. In the example below, the method `orThrow` can be invoked on any value, but it unpacks nullable types into non-nullable ones.

```
fun <T> T.orThrow(): T & Any = this ?: throw Error()

fun main() {
    val a: Int? = if (Random.nextBoolean()) 42 else null
    val b: Int = a.orThrow()
    val c: Int = b.orThrow()
    println(b)
}
```

Generic interfaces

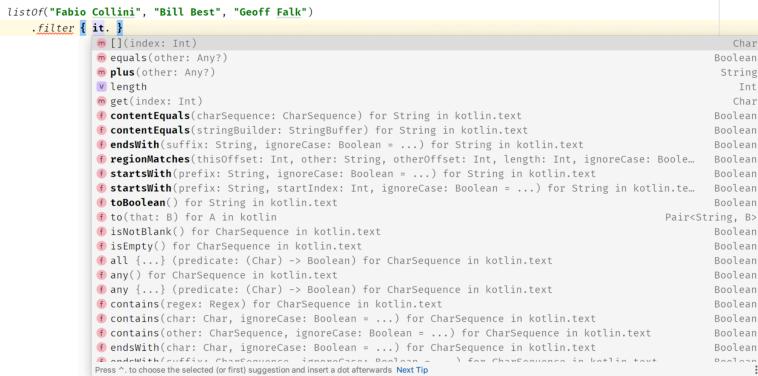
Interfaces can also be generic, which has similar consequences as for classes: the specified type parameters can be used inside the interface body as types for properties, parameters, and result types. A good example is the `List` interface.

```
interface List<out E> : Collection<E> {
    override val size: Int
    override fun isEmpty(): Boolean
    override fun contains(element: @UnsafeVariance E): Boolean
    override fun iterator(): Iterator<E>
    override fun containsAll(
        elements: Collection<@UnsafeVariance E>
    ): Boolean
    operator fun get(index: Int): E
    fun indexOf(element: @UnsafeVariance E): Int
    fun lastIndexOf(element: @UnsafeVariance E): Int
    fun listIterator(): ListIterator<E>
    fun listIterator(index: Int): ListIterator<E>
    fun subList(fromIndex: Int, toIndex: Int): List<E>
}
```

The `out` modifier and the `UnsafeVariance` annotation will be explained in the book *Advanced Kotlin*.



For `List<String>` type, methods like `contains` expect an argument of type `String`, and methods like `get` declare `String` as the result type.



For `List<String>`, methods like `filter` can infer `String` as a lambda parameter.

Generic interfaces are inherited by classes. Let's say that we have a class `Dog` that inherits from `Consumer<DogFood>`, as shown in the snippet below. The interface `Consumer` expects a method `consume` with the type parameter `T`. This means our `Dog` must override the `consume` method with an argument of type `DogFood`. It must be `DogFood` because we implement the `Consumer<DogFood>` type, and the parameter type in the interface `Consumer` must match the type argument `DogFood`. Now, when you have an instance of `Dog`, you can up-cast it to `Consumer<DogFood>`.

```
interface Consumer<T> {
    fun consume(value: T)
}

class DogFood

class Dog : Consumer<DogFood> {
    override fun consume(value: DogFood) {
        println("Mlask mlask")
    }
}

fun main() {
    val dog: Dog = Dog()
    val consumer: Consumer<DogFood> = dog
}
```

Type parameters and inheritance

Classes can inherit from open generic classes or implement generic interfaces; however, in both cases they must explicitly specify the type argument. Consider the snippet below. Class A inherits from C<Int> and implements I<String>.

```
open class C<T>
interface I<T>
class A : C<Int>(), I<String>

fun main() {
    val a = A()
    val c: C<Int> = a
    val i: I<String> = a
}
```

It is actually quite common that a non-generic class inherits from a generic one. Consider `MessageListAdapter` presented below, which inherits from `ArrayAdapter<String>`.

```
class MessageListAdapter(
    context: Context,
    val values: List<ClaimMessage>
) : ArrayAdapter<String>(
    context,
    R.layout.row_messages,
    values.map { it.title }.toTypedArray()
) {

    fun getView(
        position: Int,
        convertView: View?,
        parent: ViewGroup?
    ): View {
        // ...
    }
}
```

An even more common case is when one generic class or interface inherits from another generic class or interface and uses its type parameter as a type argument of the class it inherits from. In the snippet below, the class `A` is generic and uses its type parameter `T` as an argument for both `C` and `I`. This means that if you create `A<Int>`, you will be able to up-cast it to `C<Int>` or `I<Int>`. However, if you create `A<String>`, you will be able to up-cast it to `C<String>` or to `I<String>`.

```
open class C<T>
interface I<T>
class A<T> : C<T>(), I<T>

fun main() {
    val a: A<Int> = A<Int>()
    val c1: C<Int> = a
    val i1: I<Int> = a

    val a1: A<String> = A<String>()
    val c2: C<String> = a1
    val i2: I<String> = a1
}
```

A good example is the collection hierarchy. An object of type `MutableList<Int>` implements `List<Int>`, which implements `Collection<Int>`, which implements `Iterable<Int>`.

```
interface Iterable<out T> {
    // ...
}

interface MutableIterable<out T> : Iterable<T> {
    // ...
}

interface Collection<out E> : Iterable<E> {
    // ...
}

interface MutableCollection<E> : Collection<E>,
    MutableIterable<E> {
```

```
// ...
}
interface List<out E> : Collection<E> {
    // ...
}
interface MutableList<E> : List<E>, MutableCollection<E> {
    // ...
}
```

However, a class does not need to use its type parameter when inheriting from a generic class or implementing a generic interface. Type parameters of parent and child classes are independent of one another and should not be confused, even if they have the same name.

```
open class C<T>
interface I<T>
class A<T> : C<Int>(), I<String>

fun main() {
    val a1: A<Double> = A<Double>()
    val c1: C<Int> = a1
    val i1: I<String> = a1
}
```

Type erasure

Generic types were added to Java for developers' convenience, but they were never built into the JVM platform. All type arguments are lost when we compile Kotlin to JVM bytecode⁸⁵. Under the hood, this means that `List<String>` becomes `List`, and `emptyList<Double>` becomes

⁸⁵I use JVM as a reference because it is the most popular target for Kotlin, but also because it was the first one, so many Kotlin mechanisms were designed for it. However, regarding a lack of support for type arguments, other platforms are not better. For example, JavaScript does not support types at all.

`emptyList`. The process of losing type arguments is known as **type erasure**. Due to this process, type parameters have some limitations compared to regular types. You cannot use them for `is` checks; you cannot reference them⁸⁶; and you cannot use them as reified type arguments⁸⁷.

```
import kotlin.reflect.typeOf

fun <T> example(a: Any) {
    val check = a is T // ERROR
    val ref = T::class // ERROR
    val type = typeOf<T>() // ERROR
}
```

However, Kotlin can overcome these limitations thanks to the use of inline functions with reified type arguments. This topic is covered in depth in the chapter *Inline functions* in the book *Functional Kotlin*.

```
import kotlin.reflect.typeOf

inline fun <reified T> example(a: Any) {
    val check = a is T
    val ref = T::class
    val type = typeOf<T>()
}
```

Generic constraints

An important feature of type parameters is that they can be constrained to be a subtype of a concrete type. We set a constraint by placing a supertype after a colon. For instance, let's say that you implement the `maxOf` function, which returns

⁸⁶Class and type references are explained in the book *Advanced Kotlin*.

⁸⁷Reified type arguments are explained in the book *Functional Kotlin*.

the biggest of its arguments. To find the biggest one, the arguments need to be comparable. So, next to the type parameter, we can specify that we accept only types that are a subtype of Comparable<T>.

```
import java.math.BigDecimal

fun <T : Comparable<T>> maxOf(a: T, b: T): T {
    return if (a >= b) a else b
}

fun main() {
    val m = maxOf(BigDecimal("10.00"), BigDecimal("11.00"))
    println(m) // 11.00

    class A
    maxOf(A(), A()) // Compilation error,
    // A is not Comparable<A>
}
```

Type parameter constraints are also used for generic classes. Consider the `ListAdapter` class below, which expects a type argument that is a subtype of `ItemAdapter`.

```
class ListAdapter<T : ItemAdapter>(/*...*/) { /*...*/ }
```

An important result of having a constraint is that instances of this type can use all the methods offered by this type. In this way, when `T` is constrained as a subtype of `Iterable<Int>`, we know that we can iterate over an instance of type `T`, and that elements returned by the iterator will be of type `Int`. When we are constrained to `Comparable<T>`, we know that this type can be compared with another instance of the same type. Another popular choice for a constraint is `Any`, which means that a type can be any non-nullable type.

In rare cases in which we might need to set more than one upper bound, we can use `where` to set more constraints. We add it after the class or function name, and we use it to specify more than one generic constraint for a single type.

```
interface Animal {
    fun feed()
}

interface GoodTempered {
    fun pet()
}

fun <T> pet(animal: T) where T : Animal, T : GoodTempered {
    animal.pet()
    animal.feed()
}

class Cookie : Animal, GoodTempered {
    override fun pet() {
        // ...
    }
    override fun feed() {
        // ...
    }
}
class Cujo : Animal {
    override fun feed() {
        // ...
    }
}

fun main() {
    pet(Cookie()) // OK
    pet(Cujo()) //COMPILATION ERROR, Cujo is not GoodTempered
}
```

Star projection

In some cases, we don't want to specify a concrete type argument for a type. In these scenarios, we can use a star projection `*`, which accepts any type. There are two situations where this is useful. The first is when you check if a variable is a list. In this case, you should use the `is List<*>` check.

Star projection should be used in such a case because of type erasure. If you used `List<Int>`, it would be compiled to `List` under the hood anyway. This means a list of strings would pass the `is List<Int>` check. Such a check would be confusing and is illegal in Kotlin. You must use `is List<*>` instead.

```
fun main() {
    val list = listOf("A", "B")
    println(list is List<*>) // true
    println(list is List<Int>) // Compilation error
}
```

Star projection can also be used for properties or parameters. You can use `List<*>` when you want to express that you want a list, no matter what the type of its elements. When you get elements from such a list, they are of type `Any?`, which is the supertype of all the types.

```
fun printSize(list: List<*>) {
    println(list.size)
}

fun printList(list: List<*>) {
    for (e in list) { // the type of e is Any?
        println(e)
    }
}
```

Star projection should not be confused with the `Any?` type argument. To see this, let's compare `MutableList<Any?>` and `MutableList<*>`. Both of these types declare `Any?` as generic result types. However, when elements are added, `MutableList<Any?>` accepts anything (`Any?`), but `MutableList<*>` accepts `Nothing`, so it does not accept any values.

```
fun main() {
    val l1: MutableList<Any?> = mutableListOf("A")
    val r1 = l1.first() // the type of r1 is Any?
    l1.add("B") // the expected argument type is Any?

    val l2: MutableList<*> = mutableListOf("A")
    val r2 = l2.first() // the type of r2 is Any?
    l2.add("B") // ERROR,
    // the expected argument type is Nothing,
    // so there is no value that might be used as an argument
}
```

When a star projection is used as an argument, it will be treated as `Any?` in all the out-positions (result types), and it will be treated as `Nothing` in all the in-positions (parameter types).

Summary

For many developers, generics seem so hard and scary, but they are actually quite simple and intuitive. We can make an element generic by specifying its type parameter (or parameters). Such a type parameter can be used inside this element. This mechanism lets us generalize algorithms and classes so that they can be used with different types. It is good to understand how generics work, which is why this chapter has presented nearly all aspects of this mechanism. However, there are a few more, and we will get back to this topic in the book *Advanced Kotlin*, where we still need to discuss variance modifiers (`out` and `in`).

Final words

So, you've reached the end of this book. Congratulations! In this book, we've covered a lot of essential topics, including:

- variables, values, and types,
- conditional statements,
- functions,
- for-loops,
- support for nullability,
- classes, interfaces, and inheritance,
- object expressions and declarations,
- data, sealed, enum, and annotation classes,
- exceptions,
- extension functions,
- collections,
- operator overloading,
- the type system,
- generics.

Still, there is a lot more to learn about Kotlin. The continuation of this book is called *Functional Kotlin*, which focuses on Kotlin's functional features, including:

- function types,
- anonymous functions,
- lambda expressions,
- function references,
- functional interfaces,
- collection processing functions,
- sequences,
- DSL usage and creation,
- scope functions,
- the essentials of the Arrow library.

However, the knowledge from this book is substantial and should be enough for developing some Kotlin projects. I hope you will use it well, however you decide to continue your adventure with Kotlin.