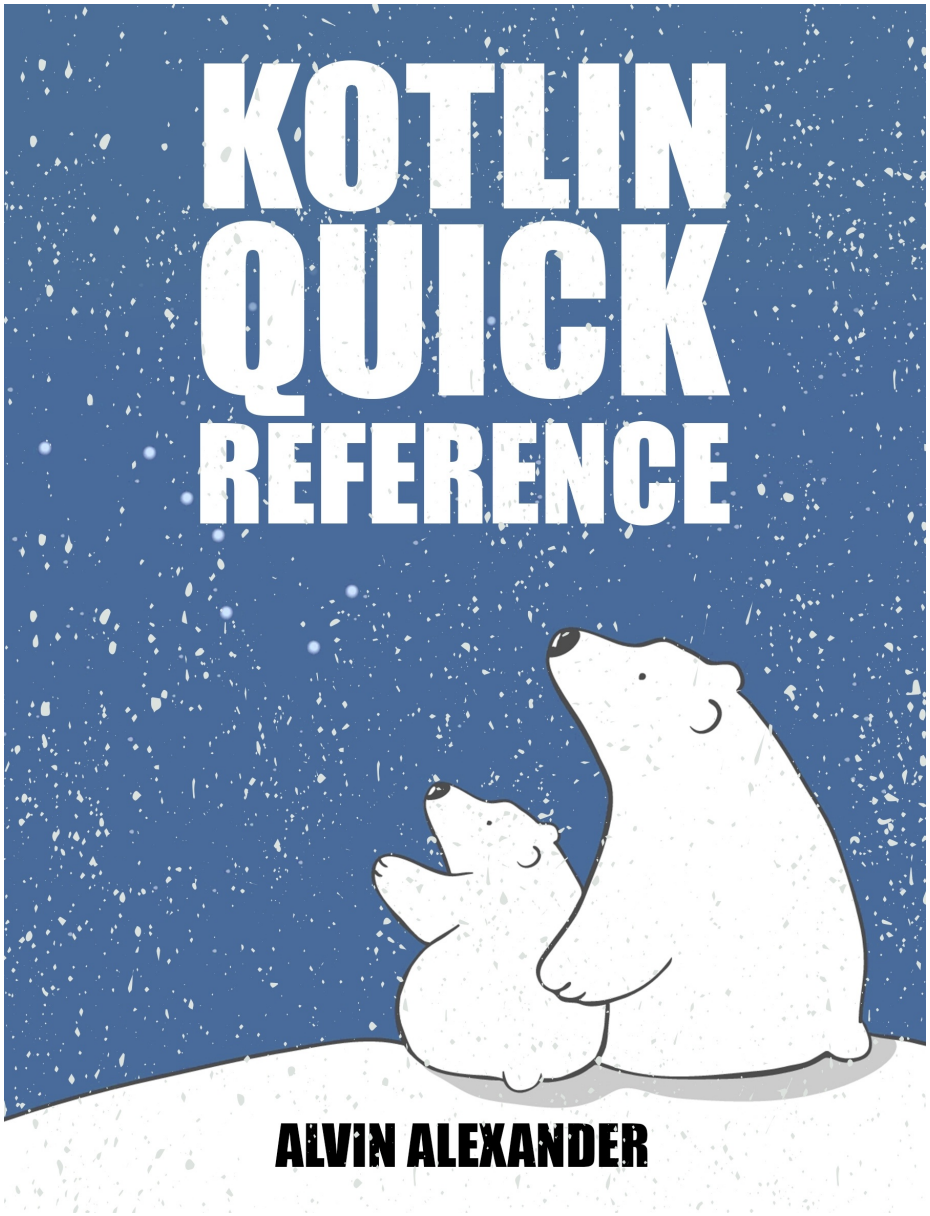
KOTLIN QUICK REFERENCE

ALVIN ALEXANDER

Table of Contents



Kotlin Quick Reference

Introduction

Kotlin Quick Reference is intended to provide a quick reference to the Kotlin programming language. Each chapter in the book demonstrates Kotlin syntax and provides examples to explain the chapter's topic. This is done in a brisk manner, with as few words as necessary.

Audience

Because this book provides only a quick reference to the Kotlin language, it's intended for developers who have experience in other programming languages and just need a fast reference to Kotlin features. If you need a more complete book, Kotlin in Action is an excellent resource.

Preface

Release information

This is Release 0.1 of Kotlin Quick Reference.

Loving Scala ... and Kotlin

Since 2011 I've been using the Scala programming language. I fell in love with as soon as I saw it, so much so that I wrote three books about it:

- Scala Cookbook
- Functional Programming, Simplified
- Hello, Scala

The *only* problem I have with Scala is that I enjoy developing Android applications, and with the exception of the Scala on Android project — which hasn't had a new release since February 24, 2017 — there isn't a simple way to create Android applications with Scala. So when I looked into Kotlin and saw that it was very similar to Scala, well, I was pretty happy.

What I love about Kotlin

What I love about both Kotlin (and Scala) can be summed up this way:

- The syntax is as elegant and concise as Ruby
- Everything feels dynamic, but it's statically typed
- Source code compiles to class files that run on the JVM
- You can use all of the thousands of Java libraries in existence
- Just as I've seen with Scala, the Kotlin creators state that as a rough estimate, Kotlin requires about 40% fewer lines of code than Java
- Developing Android applications with Kotlin is just as easy (easier!) as creating Android applications with Java

Goal of this book

While there are now many good Kotlin books available, and the documentation on the Kotlin website is excellent, I've found that what I want is a *quick reference* to the Kotlin language. Having used multiple programming languages before — including Scala — I don't need a lot of discussion about a new language, I mostly just need to see the language's syntax and some good examples.

Therefore, my goal in this book is to provide a quick reference to the Kotlin language — light on words, and heavy on demonstrating syntax and examples.

The history of this book

This book originally started as a light introduction to the Kotlin language, similar to my own "Hello, Scala" book, whose goal is to help programmers learn Scala fast. However, as I was working on it I realized that what I wanted was a quick reference, similar in style to the O'Reilly "Nutshell" books, or the book, Scala for the Impatient.

Therefore, I started converting my original “Hello, Kotlin” book into this Kotlin Quick Reference. Because of that, and because this is also a very early release of the book, some of the writing style is inconsistent. Some lessons — especially the early lessons — are written in a “Hello, Kotlin” tutorial style, while others are written in a “Nutshell” style. My goal is that in the long term all lessons will be written in the Nutshell style, with lots of source code examples and few words.

Alvin Alexander
alvinalexander.com

Getting Started With Kotlin

In this book I assume that you're familiar with at least one other programming language like Java or Scala, so I don't spend much time on programming basics. That is, I assume that you've seen things like for-loops, classes, and methods before, so I don't try to explain object-oriented or functional programming, I generally only write, "This is how you create a class in Kotlin," that sort of thing.

That being said, if you're new to Kotlin, there are a few good things to know before you jump in.

Download Kotlin

To run the examples in this book you'll need to download the Kotlin compiler and runtime environment. Visit kotlinlang.org for information about how to use Kotlin from the command line, or in the IntelliJ IDEA, Android Studio, and Eclipse IDEs.

In this book I assume that you have the Java SDK and Kotlin command line tools installed.

Comments

Comments in Kotlin are just like comments in Java (and many other languages):

```
// a single line comment

/*
 * a multiline comment
 */

/**
 * also a multiline comment
 */
```

Naming conventions

Kotlin naming conventions follow the same "camel case" style as Java and Scala:

- Class names: `Person`, `StoreEmployee`
- Variable names: `name`, `firstName`
- Method names: `convertToInt`, `toUpper`

Kotlin source files

Kotlin source code files end with the *.kt* filename extension.

Coding style

A few notes about the coding style in this book:

- I indent source code lines with four spaces, but most people seem to use two spaces. I find that four spaces makes code easier to read.

- In this book I use `val` variables in all places *unless* the feature I'm showing specifically requires the use of `var`. It's a best practice to always use `val` unless there's a good reason not to.

The Kotlin Programming Language

Kotlin is a modern programming language created by JetBrains, the makers of the IntelliJ IDEA IDE (and other products). Kotlin first appeared in 2011, was open-sourced in 2012, and is now used by companies throughout the world, including Square, Pinterest, Basecamp, Evernote, and many more.

Here are a few nuggets about Kotlin:

- Per the Kotlin Language Documentation, it's design is influenced by Java, C#, JavaScript, Scala and Groovy.
- It's a high-level language.
- It's statically typed.
- It has a sophisticated type inference system.
- It's syntax is concise but still readable. From the Kotlin Language Documentation: "Rough estimates indicate approximately a 40% cut in the number of lines of code (compared to Java)."
- It's a pure object-oriented programming (OOP) language. Every variable is an object, and every "operator" is a method.
- It can also be used as a functional programming (FP) language, so functions are also variables, and you can pass them into other functions. You can write your code using OOP, FP, or combine them in a hybrid style.
- Kotlin source code compiles to ".class" files that run on the JVM. Kotlin lets you choose between generating Java 6 and Java 8 compatible bytecode.
- Kotlin works extremely well with the thousands of Java libraries that have been developed over the years.
- Kotlin doesn't have its own collections classes, it just provides extensions to the Java collections classes.
- Kotlin can be used to create server-side applications, GUI applications with Swing and JavaFX, and Android applications. It can also be compiled to JavaScript to create web browser client-side applications.
- There's also a Kotlin/Native project "to allow compilation for platforms where virtual machines are not desirable or possible (such as iOS or embedded targets)."
- A great thing about Kotlin is that you can be productive with it on Day 1, but it's also a deep language, so as you go along you'll keep learning and finding newer, better ways to write code.
- Of all of Kotlin's benefits, what I like best is that it lets you write concise, readable code. The time a programmer spends reading code compared to the time spent writing code is said to be at least a 10:1 ratio, so writing code that's *concise and readable* is a big deal. Because Kotlin has these attributes, programmers say that it's *expressive*.

As a historical tidbit, the name comes from *Kotlin Island*, which is near St. Petersburg, Russia, where the JetBrains team has an office. Since Java was named after the Indonesian island of Java, the team thought it would be appropriate to name their new language after an island.

Hello, World

Since the release of C Programming Language, most programming books have begun with a simple “Hello, world” example, and in keeping with tradition, here’s the source code for a Kotlin “Hello, world” example:

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

Using a text editor, save that source code in a file named *Hello.kt*.

This code is similar to Java (and Scala), but notice a few things about it:

- A Kotlin function is declared with the `fun` keyword
- Compared to Java, you only have to write `println` as opposed to `System.out.println`
- In Kotlin, functions don’t have to be inside classes, so it’s perfectly legal to put a function like this in a file by itself

A fun thing about Kotlin is that you can easily create a jar file with the `kotlinc` compiler, so run this command at your command line prompt to compile that source code and create a jar file named *Hello.jar*:

```
$ kotlinc Hello.kt -include-runtime -d Hello.jar
```

Then execute the jar file with this `java` command:

```
$ java -jar Hello.jar
Hello, world
```

Welcome to the Kotlin world!

Hello, World (Part 2)

To understand how Kotlin works, let's take a look at that *Hello.kt* file again:

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

This time, rather than creating an executable jar file, just compile the code like this:

```
$ kotlinc Hello.kt
```

That command creates a file named *HelloKt.class*. Since this is a normal JVM class file, use the `javap` command to disassemble it and see what's inside:

```
$ javap HelloKt.class

Compiled from "Hello.kt"
public final class HelloKt {
  public static final void main(java.lang.String[]);
}
```

As shown, Kotlin creates a class named `HelloKt`, and it contains a normal Java `public static void main` method. Kotlin creates the class with the name `HelloKt` because we didn't supply a class name. The class file name comes from the filename — *Hello.kt* becomes `HelloKt`.

A little more fun

Before we move on, let's have a little more fun. Save this source code to a file named *HelloYou.kt*:

```
fun main(args: Array<String>) {
    if (args.size == 0)
        println("Hello, you")
    else
        println("Hello, ${args[0]}")
}
```

I hope you can see how it works:

- if/else statements work just like Java.
- Kotlin supports String Templates, which are similar to String Interpolation in languages like Scala, Groovy, and Ruby, so `${args[0]}` prints the first command line argument.

Now compile that file and create a new executable jar file:

```
$ kotlinc HelloYou.kt -include-runtime -d HelloYou.jar
```

Then run it with and without a command line argument:

```
$ java -jar HelloYou.jar
Hello, you

$ java -jar HelloYou.jar Alvin
```

```
Hello, Alvin
```

Extra credit

To have a little more fun with this example, run this Java `jar` command on *HelloYou.jar*:

```
$ jar tvf HelloYou.jar
```

If you know Java and how the JVM works, you can guess that the initial output of that command looks like this:

```
  79 Wed Aug 01 13:54:12 MDT 2018 META-INF/MANIFEST.MF
1249 Wed Aug 01 13:54:12 MDT 2018 HelloYouKt.class
```

The rest of the output is too long to include here — 654 lines total, to be precise — but I encourage you to run that command to get an idea of what's needed to create an executable jar file.

The Kotlin REPL

The Kotlin REPL ("Read-Evaluate-Print-Loop") is a command-line interpreter that you use as a "playground" area to test your Kotlin code. To start a REPL session, just type `kotlinc` at your operating system command line, and you'll see this:

```
> kotlinc
Welcome to Kotlin version 1.2.51 (JRE 1.8.0_181-b13)
Type :help for help, :quit for quit
>>> _
```

> The prompt for the Kotlin REPL is `>>>`, but because I don't like that I only show one `>` character for the prompt in this book.

Because the REPL is a command-line interpreter, it just sits there waiting for you to type something. Once you're in the REPL, you can type Kotlin expressions to see how they work:

```
> 1 + 1
2

> 4 / 2
2

> 4 % 2
0
```

As those examples show, when you type simple expressions, the Kotlin REPL shows the result of the expression on the line after the prompt. For other expressions you may have to type the variable name you created to see its value:

```
> val x = 1 + 1

> x
2
```

> You can also try Kotlin online at try.kotlinlang.org/

Take the REPL for a spin

You're going to use the Kotlin REPL a lot in this book, so go ahead and start it (with `kotlinc`) and experiment with it. Here are a few expressions you can try to see how it all works:

```
1 + 1
4 / 2
5 / 2
4 % 2
5 % 2
5 / 2.0
1 + 2 * 3
(1 + 2) * 3
val name = "John Doe"
"hello"[0]
"hello"[1]
"hello, world".take(5)
println("hi")
if (2 > 1) println("greater") else println("lesser")
for (i in 1..3) println(i)
```

```
listOf(1,2,3).forEach { println(it) }
```

As a little more advanced exercise, here's how to define a class named `Person`, create an instance of it, and then show the value of the `name` field:

```
class Person(var name: String)
val p = Person("Kim")
p.name
```

Notice that you don't need the `new` keyword when creating an instance of a class.

Use `javaClass` to see an object's type

To see an object's type in the REPL, call `.javaClass` on the instance:

```
> "foo".javaClass
class java.lang.String

> 1.javaClass
int

> 1.0.javaClass
double
```

Here's a map:

```
// paste this into the repl
val map = mapOf(
    1 to "one",
    2 to "two"
)

> map.javaClass
class java.util.LinkedHashMap
```

Here's a list:

```
> listOf(1,2,3).javaClass
class java.util.Arrays$ArrayList
```

The following examples on the result of `intArrayOf` show other calls you can make after `javaClass`:

```
> val x = intArrayOf(1,2,3)

> println(x.javaClass.name)
[I

> println(x.javaClass.kotlin)
class kotlin.IntArray

> println(x.javaClass.kotlin.qualifiedName)
kotlin.IntArray
```

Two Types of Variables

Kotlin has two types of variable declarations:

- `val` creates an *immutable* variable (like `final` in Java)
- `var` creates a *mutable* variable

Examples:

```
val s = "hello"   // immutable
var i = 42        // mutable

val p = Person("Hala")
```

Notes:

- Kotlin can usually infer the variable's data type from the code on the right side of the `=` sign.
- This is considered an *implicit* form.
- You can also *explicitly* declare the variable type if you prefer:

```
val s: String = "hello"
var i: Int = 42
```

- The implicit form is generally preferred
- I use the explicit form when I don't think the type is obvious; code is easier to maintain that way

The difference between `val` and `var`

The REPL shows what happens when you try to reassign a `val` field:

```
> val a = 'a'

> a = 'b'
error: val cannot be reassigned
a = 'b'
^
```

That fails with a "val cannot be reassigned" error, as expected. Conversely, you can reassign a `var`:

```
> var a = 'a'

> a = 'b'

> a
b
```

The general rule is that you should always use a `val` field unless there's a good reason not to. This simple rule has several benefits:

- It makes your intention obvious: you don't want this field to be reassigned
- It makes your code more like algebra
- If you ever want to go there, it helps get you started down the path to *functional programming*, where *all* fields are immutable

“Hello, world” with a `val` field

Here’s a “Hello, world” app with a `val` field:

```
fun main(args: Array<String>) {
    val s = "Hello, world"
    println(s)
}
```

As before:

- Save that code in a file named *HelloVal.kt*
- Compile it with `kotlinc HelloVal.kt -include-runtime -d HelloVal.jar`
- Run it with `java -jar HelloVal.jar`

Deferred initialization of `val`

Per Kotlin in Action, “a `val` variable must be initialized exactly once during the execution of the block where it’s defined ... but you can initialize it with different values depending on some condition.” Examples:

```
val name: String
val num: Int

val r = (1..10).shuffled().first()

// assign `name` and `num`
name = if (r % 2 == 0) "Alvin" else "Alexander"
num = r

println("name = $name, num = $num")
```

A note about `val` fields in the REPL

You can’t reassign a `val` field in the REPL:

```
> val age = 18

> age = 19
error: val cannot be reassigned
age = 19
^
```

However, you can do this in the REPL:

```
> val age = 18

> val age = 19
```

I thought I’d mention that because I didn’t want you to see it one day and think, “Hey, Al said `val` fields couldn’t be reassigned.” They can be reassigned like that, but only in the REPL.

Declaring the Type is Optional

As I showed in the previous lesson, when you create a new variable in Kotlin you can *explicitly* declare its type, like this:

```
val count: Int = 1
val name: String = "Alvin"
```

However, you can generally leave the type off and Kotlin can infer it for you:

```
val count = 1
val name = "Alvin"
```

In most cases your code is easier to read when you leave the type off, so this implicit form is preferred.

The explicit form feels verbose

One thing you'll find is that the explicit form feels verbose. For instance, in this example it's obvious that the data type is `Person`, so there's no need to declare the type on the left side of the expression:

```
val p = Person("Candy")
```

By contrast, when you put the type next to the variable name, the code feels unnecessarily verbose:

```
val p: Person = Person("Leo")
```

Summary:

```
val p = Person("Candy")          // preferred
val p: Person = Person("Candy")  // unnecessarily verbose
```

Use the explicit form when you need to be clear

One place where you'll want to show the data type is when you want to be clear about what you're creating. That is, if you don't explicitly declare the data type, the compiler may make a wrong assumption about what you want to create. Some examples of this are when you want to create numbers with specific data types. I show this in the next lesson.

Built-In Data Types

Kotlin comes with the standard numeric data types you'd expect, and in Kotlin all of these data types are full-blown objects — not primitive data types.

How to declare variables of the basic numeric types:

```
val b: Byte = 1
val i: Int = 1
val l: Long = 1
val s: Short = 1
val d: Double = 2.0
val f: Float = 3.0f
```

In the first four examples, if you don't explicitly specify a type, the number `1` will default to an `Int`, so if you want one of the other data types — `Byte`, `Long`, or `Short` — you need to explicitly declare those types, as shown. Numbers with a decimal (like 2.0) will default to a `Double`, so if you want a `Float` you need to declare a `Float`, as shown in the last example. You can also declare `Long` and `Float` types like this:

```
val l = 1L
val f = 3.0f
```

Because `Int` and `Double` are the default numeric types, you typically create them without explicitly declaring the data type:

```
val i = 123   // defaults to Int
val x = 1.0   // defaults to Double
```

All of those data types have the same data ranges as their Java equivalents:

Type	Bit width
Byte	8
Short	16
Int	32
Long	64
Float	32
Double	64

(For more information on those, see my article, JVM bit sizes and ranges.)

BigInteger and BigDecimal

In Kotlin you can use the *java.math.BigInteger* class:

```
> import java.math.BigInteger
> val x = BigInteger("1")
```

Kotlin also has convenient extension functions to help you convert other data types to `BigInteger`:

```
> val y = 42.toBigInteger()
> val y = 42L.toBigInteger()
```

Kotlin lets you use the Java `BigDecimal` class in similar ways:

```
> import java.math.BigDecimal
> val x = BigDecimal("1.0")
> 1.0.toBigDecimal()
```

See these links for more information:

- Kotlin BigInteger extension functions
- Kotlin BigDecimal extension functions

String, Char, and Boolean

Kotlin also has `String`, `Char`, and `Boolean` data types, which I always declare with the implicit form:

```
val name = "Bill"
val c = 'c'
val b = true
```

Strings

This lesson shows:

- String templates
- Multiline strings
- How to convert a list or array to a `String`

String templates

Kotlin has a nice, Ruby- and Scala-like way to merge multiple strings known as String Templates. Given these three variables:

```
val firstName = "John"
val mi = 'C'
val lastName = "Doe"
```

you can append them together like this:

```
val name = "$firstName $mi $lastName"
```

This creates a very readable way to print multiple strings:

```
println("Name: $firstName $mi $lastName")
```

You can also include complete expressions inside strings:

```
> val x = 2
> val y = 3
> println("$x times $y is ${x * y}.")
2 times 3 is 6.
```

Multiline strings

You can create multiline strings by including the string inside three parentheses:

```
val speech = """Four score and
               seven years ago
               our fathers ..."""
```

One drawback of this basic approach is that lines after the first line are indented, as you can see in the REPL:

```
> speech
Four score and
               seven years ago
               our fathers ...
```

A simple way to fix this problem is to put a `|` symbol in front of all lines after the first line, and call the `trimMargin` function after the string:

```
val speech = """Four score and
               |seven years ago
               |our fathers ...""".trimMargin()
```

The REPL shows that when you do this, all of the lines are left-justified:

```
> speech
Four score and
seven years ago
our fathers ...
```

How to convert a list to a String

How to convert a list or array to a `String`:

```
val nums = listOf(1,2,3,4,5)

> nums.joinToString()
1, 2, 3, 4, 5

> nums.joinToString(
    separator = ", ",
    prefix = "[",
    postfix = "]",
    limit = 3,
    truncated = "there's more ..."
)
[1, 2, 3, there's more ...]
```

Another example:

```
val words = arrayOf("Al", "was", "here")

> words.joinToString()
Al, was, here

> words.joinToString(separator = " ")
Al was here
```

I/O With Kotlin

To get ready to show `for` loops, `if` expressions, and other Kotlin constructs, let's take a look at how to handle I/O with Kotlin.

Writing to STDOUT and STDERR

Write to standard out (STDOUT) using `println`:

```
println("Hello, world")          // includes newline
print("Hello without newline")   // no newline character
```

Because `println` is so commonly used, there's no need to import it.

Writing to STDERR:

```
System.err.println("yikes, an error happened")
```

Reading command-line input

A simple way to read command-line (console) input is with the `readLine()` function:

```
print("Enter your name: ")
val name = readLine()
```

`readLine()` provides a simple way to read input. For more complicated needs you can also use the *java.util.Scanner* class, as shown in this example:

```
import java.util.Scanner
val scanner = Scanner(System.`in`)
print("Enter an int: ")
val i: Int = scanner.nextInt()
println("i = $i")
```

Just be careful with the `Scanner` class. If you're looking for an `Int` and the user enters something else, you'll end up with a `InputMismatchException`:

```
>>> val i: Int = scanner.nextInt()
1.1
java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
```

> I write more about the `Scanner` class in my article, How to prompt users for input in Scala shell scripts.

Reading text files

There are a number of ways to read text files in Kotlin. While this approach isn't recommended for large text files, it's a simple way to read small-ish text files into a `List<String>`:

```
import java.io.File
fun readFile(filename: String): List<String> = File(filename).readLines()
```

Here's how you use that function to read the */etc/passwd* file:

```
val lines = readFile("/etc/passwd")
```

And here are two ways to print all of those lines to STDOUT:

```
lines.forEach{ println(it) }
lines.forEach{ line -> println(line) }
```

Other ways to read text files

Here are a few other ways to read text files in Kotlin:

```
fun readFile(filename: String): List<String> = File(filename).readLines()

fun readFile(filename: String): List<String> = File(filename).bufferedReader().readLines()

fun readFile(filename: String): List<String> = File(filename).useLines { it.toList() }

fun readFile(filename: String): String = File(filename).inputStream().readBytes().toString(Charsets.UTF_8)

val text = File("/etc/passwd").bufferedReader().use { it.readText() }
```

The file-reading function signatures look like this:

```
// Do not use this function for huge files.
fun File.readLines(
    charset: Charset = Charsets.UTF_8
): List<String>

fun File.bufferedReader(
    charset: Charset = Charsets.UTF_8,
    bufferSize: Int = DEFAULT_BUFFER_SIZE
): BufferedReader

fun File.reader(
    charset: Charset = Charsets.UTF_8
): InputStreamReader

fun <T> File.useLines(
    charset: Charset = Charsets.UTF_8,
    block: (Sequence<String>) -> T
): T

// This method is not recommended on huge files. It has an internal limitation of 2 GB byte array size.
fun File.readBytes(): ByteArray

// This method is not recommended on huge files. It has an internal limitation of 2 GB file size.
fun File.readText(charset: Charset = Charsets.UTF_8): String
```

See the Kotlin java.io.File documentation for more information and caveats about the methods shown (`readLines` , `useLines` , etc.).

Writing text files

There are several ways to write text files in Kotlin. Here's a simple approach:

```
File(filename).writeText(string)
```

That approach default to the UTF-8 character set. You can also specify the `Charset` when using `writeText`:

```
File(filename).writeText(string, Charset.forName("UTF-16"))
```

Other ways to write files

There are other ways to write to files in Kotlin. Here are some examples:

```
File(filename).writeBytes(byteArray)
File(filename).printWriter().use { out -> out.println(string) }
File(filename).bufferedWriter().use { out -> out.write(string) }
```

The file-writing function signatures look like this:

```
fun File.writeText(
    text: String,
    charset: Charset = Charsets.UTF_8
)

fun File.writeBytes(array: ByteArray)

fun File.printWriter(
    charset: Charset = Charsets.UTF_8
): PrintWriter

fun File.bufferedWriter(
    charset: Charset = Charsets.UTF_8,
    bufferSize: Int = DEFAULT_BUFFER_SIZE
): BufferedWriter
```

See the Kotlin java.io.File documentation for more information about the methods shown.

Kotlin Control Structures

Kotlin has the basic control structures you'd expect to find in a programming language, including:

- if/then/else-if/else
- `for` loops
- try/catch/finally
- `while` and `do..while` loops

It also has unique constructs, including:

- `when` expressions, which are like improved Java `switch` statements

I'll demonstrate those in the following lessons.

if/then/else

`if` statements in Kotlin are just like Java, with one exception: they are an expression, so they always return a result.

Basic syntax

A basic Kotlin `if` statement:

```
if (a == b) doSomething()
```

You can also write that statement like this:

```
if (a == b) {
    doSomething()
}
```

if/else

The `if` / `else` construct:

```
if (a == b) {
    doSomething()
} else {
    doSomethingElse()
}
```

if/else-if/else

The complete Kotlin if/else-if/else expression looks like this:

```
if (test1) {
    doX()
} else if (test2) {
    doY()
} else {
    doZ()
}
```

`if` expressions always return a result

The `if` construct always returns a result, meaning that you can use it as an expression. This means that you can assign its result to a variable:

```
val minValue = if (a < b) a else b
```

This means that Kotlin doesn't require a special ternary operator.

Kotlin `for` Loops

- `for` loops are similar to Java, but the syntax is different
- Unlike in Scala, Kotlin `for` loops are not expressions

Basic syntax

Kotlin `for` loops have this basic syntax:

```
for (e in elements) { // do something with e ...}
```

For example, given a list:

```
val nums = listOf(1,2,3)
```

here's a single-line algorithm:

```
for (n in nums) println(n)
```

The REPL shows how it works:

```
>>> for (n in nums) println(n)
1
2
3
```

You can also use multiple lines of code in your algorithm:

```
for (n in 1..3) {
    println(n)
}
```

Here are a couple of other examples:

```
// directly on listOf
for (n in listOf(1,2,3)) println(n)

// 1..3 creates a range
for (n in 1..3) println(n)
```

Using `for` with Maps

You can also use a `for` loop with a Kotlin `Map` (which is similar to a Java `HashMap`). This is how you create a `Map` of movie names and ratings:

```
val ratings = mapOf(
    "Lady in the Water"  to 3.0,
    "Snakes on a Plane"  to 4.0,
    "You, Me and Dupree" to 3.5
)
```

Given that map, you can print the movie names and ratings with this `for` loop:

```
for ((name,rating) in ratings) println("Movie: $name, Rating: $rating")
```

Here's what that looks like in the REPL:

```
>>> for ((name,rating) in ratings) println("Movie: $name, Rating: $rating")
Movie: Lady in the Water, Rating: 3.0
Movie: Snakes on a Plane, Rating: 4.0
Movie: You, Me and Dupree, Rating: 3.5
```

In this example, `name` corresponds to each *key* in the map, and `rating` is the name I assign for each *value* in the map.

`for` is not an expression

If you're coming to Kotlin from Scala, it's important to note that `for` is NOT an expression. That means that code like this does not work:

```
// error
val x = for (n in 1..3) {
    return n * 2
}
```

while and do..while

Kotlin's `while` and `do..while` constructs are just like Java.

while

Here's `while`:

```
var i = 1

while (i < 5) {
    println(i++)
}
```

Here's its output in the REPL:

```
>>> while (i < 5) {
...     println(i++)
... }
1
2
3
4
```

do..while

Here's do..while:

```
var i = 1

do {
    println(i++)
} while (i < 0)
```

and its REPL output:

```
>>> do {
...     println(i++)
... } while (i < 0)
1
```

Notice that do..while always runs at least once because it executes its body before running the first test.

`when` Expressions

Kotlin `when` expressions are like Java `switch` statements, but you'll see in this lesson, they're much more powerful. If you're experienced with Scala, `when` is similar to Scala's `match` expression.

Replacement for `switch`

In its most basic use, `when` can be used as a replacement for a Java `switch` statement:

```
val dayAsInt = 1

when (dayAsInt) {
    1 -> println("Sunday")
    2 -> println("Monday")
    3 -> println("Tuesday")
    4 -> println("Wednesday")
    5 -> println("Thursday")
    6 -> println("Friday")
    7 -> println("Saturday")
    else -> {
        // notice you can use a block
        println("invalid day")
    }
}
```

`when` is an expression

`when` can also be used as an expression, meaning that it returns a value:

```
val dayAsInt = 1

val dayAsString = when (dayAsInt) {
    1 -> "Sunday"
    2 -> "Monday"
    3 -> "Tuesday"
    4 -> "Wednesday"
    5 -> "Thursday"
    6 -> "Friday"
    7 -> "Saturday"
    else -> "invalid day"
}

println(dayAsString)
```

Matches must be exhaustive

When `when` is used as an expression you generally must include an `else` clause. If you don't, you risk the chance of getting an error, as in this example:

```
val i = 0

val result = when (i) {
    1 -> "a little odd"
    2 -> "a little even"
}
```

That code produces this error message:

```
error: 'when' expression must be exhaustive, add necessary 'else' branch
val result = when (i) {
             ^
```

Multiple branch conditions

When you have multiple branch conditions with constants that should be handled the same way, they can be combined in one statement with a comma:

```
when (i) {
    1,2,3 -> println("got a 1, 2, or 3")
    4,5,6 -> println("got a 4, 5, or 6")
    else  -> println("something else")
}
```

Testing against ranges

You can also check a value for being in a range (`in`) or not in a range (`!in`):

```
val i = 1

when (i) {
    in 1..3  -> println("1, 2, or 3")
    !in 4..5 -> println("not a 4 or 5")
    else     -> println("something else")
}

// result: 1, 2, or 3
```

The order of the expressions is important. In this example I reverse the first two possible matches, and get a different result:

```
val i = 1

when (i) {
    !in 4..5 -> println("not a 4 or 5")
    in 1..3  -> println("1, 2, or 3")
    else     -> println("something else")
}

// result: not a 4 or 5
```

These examples also show that the `when` expression stops on the first statement that matches the given value.

`in` with `listOf`

Similar to ranges, you can also use `in` with `listOf` as the predicate condition:

```
val i = 1
when (i) {
    in listOf(1,3,5) -> println("a little odd")
    in listOf(2,4,6) -> println("a little even")
    else             -> println("something else")
}
```

Expressions as branch conditions

`when` expressions aren't limited to using only constants, you can also use any sort of predicate as a branch condition. Here are some tests with Kotlin's `is` operator:

```
val x: Any = 11.0

when (x) {
    is Boolean -> println("$x is a Boolean")
    is Double  -> println("$x is a Double")
    is String  -> println("$x is a String")
    !is String -> println("$x is not a String")
    else       -> println("$x is something else")
}
```

When you run that code as shown the result is:

```
11.0 is a Double
```

Here's an example that demonstrates how to use a function (`toUpperCase()`) in a branch condition:

```
fun isUpperCase(s: String): Boolean {
    return when (s) {
        s.toUpperCase() -> true
        else -> false
    }
}
```

The REPL shows how this works:

```
>>> isUpperCase("FOO")
true

>>> isUpperCase("foo")
false

>>> isUpperCase("Foo")
false
```

`when` without arguments

If you don't give `when` an argument, it can be used as a replacement for an if/else expression:

```
val i = 1

when {
    i < 0  -> println("less than zero")
    i == 0 -> println("zero")
    else   -> println("greater than zero")
}
```

> Key point: Notice that this example uses `when`, and not `when(i)`.

This syntax is useful for using `when` as the body of a function:

```
fun intToString(i: Int): String = when {
    i < 0  -> "a negative number!"
    i == 0 -> "0"
    else   -> "a positive number!"
}
```

Notice in this example that `when` is used as the body of the `intToString` function, and returns a `String`. If you haven't seen that approach before, it can help to see it broken down into steps:

```
fun intToString(i: Int): String {
    // convert the Int to a String
    val string = when {
        i < 0  -> "a negative number!"
        i == 0 -> "0"
        else   -> "a positive number!"
    }
    // return the String
    return string
}
```

Smart casts with `when`

When you use an `is` expression as a branch condition you can check the type of the variable passed into `when`, and call methods on it on the right-hand side of the expression. Here's an example:

```
fun getInfo (x: Any): String {
    return when (x) {
        is Int    -> "The Int plus one is ${x + 1}"
        is String -> "Got a String, length is " + x.length
        else -> "Got something else"
    }
}
```

Here's what the function looks like if you call it several times with different types:

```
println(getInfo(1))      //The Int plus one is 2
println(getInfo("foo"))  //Got a String, length is 3
println(getInfo(1L))     //Got something else
```

Even more!

While that covers most of `when`'s functionality, there are even more things you can do with it.

Kotlin try/catch/finally Expressions

Like Java, Kotlin has a try/catch/finally construct to let you catch and manage exceptions. If you've used Java, there's only one new feature here: a `try` expression is truly an expression, meaning you can assign its result to a variable. (See the "Try is an expression" section below for those details.)

Basic syntax

Kotlin's try/catch/finally syntax looks like this:

```
try {
    // some exception-throwing code
}
catch (e: SomeException) {
    // code
}
catch (e: SomeOtherException) {
    // code
}
finally {
    // optional
}
```

As shown in the comment, the `finally` clause is optional. Everything works just like Java.

Example

Because there are easier ways to read files in Kotlin, this example is a little contrived, but it shows an example of how to use `try` with multiple `catch` clauses and a `finally` clause:

```
import java.io.*

fun main(args: Array<String>) {

    var linesFromFile = listOf<String>()
    var br: BufferedReader? = null

    try {
        br = File("/etc/passwd2").bufferedReader()
        linesFromFile = br.readLines()
    }
    catch (e: IOException) {
        e.printStackTrace()
    }
    catch (e: FileNotFoundException) {
        e.printStackTrace()
    }
    finally {
        br?.close()
    }

    linesFromFile.forEach { println(">  " + it) }

}
```

Don't worry about the `?` symbols in the code for now, I just wanted to show a complete try/catch/finally example. Just know that the question marks are needed for working with null values in Kotlin.

> Or, if you want to learn about them, jump ahead to the "Nullability" lessons later in this book.

`Try` is an expression

Try in Kotlin is different than Java because Kotlin's `try` is an expression, meaning that it returns a value:

```
val s = "1"

// using try as an expression
val i = try {
    s.toInt()
}
catch (e: NumberFormatException) {
    0
}
```

That expression can be read as, "If `s` converts to an `Int`, return that integer value; otherwise, if it throws a `NumberFormatException`, return `0`."

You can write that try expression more concisely:

```
val i = try { s.toInt() } catch (e: NumberFormatException) { 0 }
```

Try that expression with different values like these to confirm for yourself that it works as expected:

```
val s = "1"
val s = "foo"
```

Classes and Objects

This section of the book covers everything related to classes and objects, including:

- Classes
- Data classes
- Objects
- The `object` keyword
- Companion objects
- More ...

Constructors and Members

In support of object-oriented programming (OOP), Kotlin provides a *class* construct. The syntax is much more concise than languages like Java and C#, but it's also still easy to use and read.

Overview

- Classes are created with the `class` keyword
- Primary constructor parameters go in the class header
  - `var` parameters are read-write, and `val` parameters are read-only
- Class instances are created *without* the `new` keyword
  - Think of it like you're calling a function
- Classes can have initializer blocks
- Classes can have properties and methods (functions)
- Classes can have nested classes and inner classes

Creating a class

Classes are created with `class` keyword:

```
class Foo
class Person constructor(var firstName: String, var lastName: String)
```

The primary constructor is part of the class header. If the primary constructor does not have any annotations or visibility modifiers, the `constructor` keyword can be omitted:

```
class Person(var firstName: String, var lastName: String)
```

I show annotations and visibility modifiers at the end of this lesson.

How to create an instance of a class

New instances of classes are created *without* the `new` keyword. You can think of them as being like function calls:

```
val f = Foo()
val p = Person("Bill", "Panner")
```

Comparison to Java

With the exception of the names of the getter and setter functions, if you're coming to Kotlin from Java, this Kotlin code:

```
class Person(var firstName: String, var lastName: String)
```

is the equivalent of this Java code:

```
public class Person {

    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

}
```

Constructor parameter visibility

Class constructor parameters are defined in the class header:

```
class Person(var firstName: String, var lastName: String)
             ---                    ---

class Person(val firstName: String, val lastName: String)
             ---                    ---
```

`var` means fields can be read-from and written-to (they are mutable); they have both a getter and a setter. `val` means the fields are read-only (they are immutable).

Accessing constructor parameters

Given this definition with `var` fields:

```
class Person(var firstName: String, var lastName: String)
             ---                    ---
```

you can create a new `Person` instance like this:

```
val p = Person("Bill", "Panner")
```

You access the `firstName` and `lastName` fields like this:

```
println("${p.firstName} ${p.lastName}")
Bill Panner
```

The REPL shows that the fields can be updated:

```
> class Person(var firstName: String, var lastName: String)
> val p = Person("Bill", "Panner")

> p.firstName = "William"
> p.lastName = "Bernheim"

> println("${p.firstName} ${p.lastName}")
William Bernheim
```

`val` fields are read-only

Had the fields been defined as `val`:

```
class Person(val firstName: String, val lastName: String)
             ---                    ---
```

you can still access them (via a getter method), but you can't change their values (their is no setter method):

```
> class Person(val firstName: String, val lastName: String)
> val p = Person("Bill", "Panner")

// can access the fields
> println(p.firstName)
Bill
> println(p.lastName)
Panner

// can't mutate the fields
> p.firstName = "William"
error: val cannot be reassigned
p.firstName = "William"
^
> p.lastName = "Bernheim"
error: val cannot be reassigned
p.lastName = "Bernheim"
^
```

Summary:

- `val` fields are read-only
- `var` fields are read-write

> Pro tip: If you use Kotlin to write OOP code, create your fields as `var` fields so you can easily mutate them. If you prefer an FP style of programming you may prefer Kotlin's *data classes*. (More on this later.)

A note about annotations and visibility modifiers

Per the Kotlin documentation, "If the constructor has annotations or visibility modifiers, the `constructor` keyword is required, and the modifiers go before it":

```
class Customer public @Inject constructor(name: String) { ... }
                               -----------
```

Visibility modifiers

Visibility modifiers are:

- private
- protected
- internal
- public

Per this kotlinlang.org link, “classes, objects, interfaces, constructors, functions, properties and their setters can have visibility modifiers.”

Here’s an example of `internal`:

```
// `internal` is visible everywhere in the same “module”
// see https://kotlinlang.org/docs/reference/visibility-modifiers.html
class TabAdapter internal constructor(fm: FragmentManager) : FragmentStatePagerAdapter(fm) {...
                 --------
```

Imports and Packages

Import and package statements are very similar to Java, with just a few additions/improvements.

Key points

Packages:

- Package statements are just like Java, but they don't have to match the directory the file is in

Import statements:

- There is no `import static` syntax
- You can rename a class when you import it
- The `import` statement is not restricted to only importing class

You can also import:

- Top-level functions and properties
- Functions and properties declared in object declarations
- Enum constants

Finally, a collection of classes and functions are imported for you by default, such as *kotlin.**

Package statements

Put package statements at the top of a file:

```
package foo.bar.baz

// the rest of your code here ...
```

The only real difference with Java is that the package name doesn't have to match the name of the directory that the file is in.

Files don't have to contain class declarations

Because a Kotlin file doesn't have to contain a class, it's perfectly legal to put one or more functions in a file:

```
package foo.bar

fun plus1(i: Int) = i + 1
fun double(i: Int) = i * 2
```

Because of the package name, you can now refer to the function `plus1` in the rest of your code as `foo.bar.plus1`. Or, as you'll see in the `import` examples that follow, you can import the function into the scope of your other code like this:

```
import foo.bar.plus1
```

Import statements

Import statements work just like Java, with only a few differences:

- There is no `import static` syntax
- You can rename a class when you import it
- The `import` statement is not restricted to only importing class

No “import static”

Kotlin doesn’t have a separate `import static` syntax. Just use a regular `import` declaration to import static methods and fields:

```
>>> import java.lang.Math.PI
>>> PI
3.141592653589793

>>> import java.lang.Math.pow
>>> pow(2.0, 2.0)
4.0
```

Renaming a class when you import it

To avoid namespace collisions you can rename a class when you import it:

```
import java.util.HashMap as JavaHashMap
```

The REPL shows how it works:

```
>>> import java.util.HashMap as JavaHashMap

>>> val map = JavaHashMap<String, String>();

>>> map.put("first_name", "Alvin")
null

>>> map
{first_name=Alvin}
```

Here’s a combination of the two techniques just shown, (a) importing a static field and (b) renaming it during the import process:

```
import java.lang.Integer.MAX_VALUE as MAX_INT
import java.lang.Long.MAX_VALUE as MAX_LONG
```

Here are the values in the REPL:

```
>>> MAX_INT
2147483647

>>> MAX_LONG
9223372036854775807
```

Things you can import

Per the official Kotlin documentation, in addition to importing classes, you can also import:

- Top-level functions and properties
- Functions and properties declared in object declarations
- Enum constants

Default imports

When you first start working with Kotlin it can be a surprise that you can use the `println` function without having to write `System.out.println` every time. This is partly due to the fact that a collection of packages are imported into every Kotlin file by default:

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.* (since 1.1)
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

The `println` function is declared in the kotlin.io package, and it's automatically made available to you.

When working with the Java Virtual Machine (JVM), these packages are also automatically imported:

- java.lang.*
- kotlin.jvm.*

When you're compiling Kotlin code to JavaScript this package is imported by default:

- kotlin.js.*

Class Members

Classes can contain:

- Constructors
- Constructor initializer blocks
- Functions
- Properties
- Nested and Inner Classes
- Object Declarations

Constructors

Basic constructor syntax:

```
class Person

class Person (name: String, age: Int) {
    // class code here ...
}
```

The `constructor` keyword is only needed if the class has annotations or visibility modifiers:

```
class Person private constructor (name: String) { ... }
class Person public @Inject constructor (name: String) { ... }
class Person @JvmOverloads constructor (name: String, age: Int) { ... }
```

Primary constructor initialization code goes in an `init` block:

```
class Person (name: String) {
    init {
        println("Person instance created")
    }
}
```

Initializer blocks

Classes can have initializer blocks that are called when the class is constructed, and those blocks can access the constructor parameters:

```
// a network socket
class Socket(var timeout: Int, var linger: Int) {
    init {
        println("Entered 'init' ...")
        println("timeout = ${timeout}, linger = ${linger}")
    }
}
```

Here's what it looks like when a `Socket` is created:

```
> val s = Socket(2000, 3000)
Entered 'init' ...
```

```
timeout = 2000, linger = 3000
```

You can have multiple initializer blocks, and the code in those blocks is effectively part of the primary constructor.

Methods (functions)

Just like Java and other languages, classes can have methods (which are referred to as functions):

```
class Person(val firstName: String, val lastName: String) {
    fun fullName() = "$firstName $lastName"
}
```

Here's a REPL example:

```
> val p = Person("Hala", "Cina")
> p.fullName()
Hala Cina
```

Properties (fields)

Just like Java and other languages, classes can have properties (fields):

```
class Pizza () {
    val toppings = mutableListOf<String>()
    fun addTopping(t: String) { toppings.add(t) }
    fun showToppings() { println(toppings) }
    // more code ...
}
```

Here's an example in the REPL:

```
> val p = Pizza()
> p.addTopping("Cheese")
> p.addTopping("Pepperoni")
> p.showToppings()
[Cheese, Pepperoni]
```

Nested and inner classes

Classes can be nested in other classes. Note that a nested class can't access a parameter in the outer class:

```
class Outer {
    private val x = 1
    class Nested {
        //fun foo() = 2 * x   //this won't compile
        fun foo() = 2
    }
}
```

In the REPL:

```
> val foo = Outer.Nested().foo()
> foo
```

```
2
```

Inner classes

Mark a class as `inner` if you need to access members of the outer class:

```
class Outer {
    private val x = 1
    inner class Inner {
        fun foo() = x * 2
    }
}
```

In the REPL:

```
> val foo = Outer().Inner().foo()
> foo
2
```

Syntax

Notice the difference between the syntax when using a nested class versus an inner class:

```
val foo = Outer.Nested().foo()
val foo = Outer().Inner().foo()
               --
```

If you try to create an inner class without first creating an instance of the outer class you'll see this error:

```
> val foo = Outer.Inner().foo()
error: constructor of inner class Inner can be called only with receiver of containing class
val foo = Outer.Inner().foo()
                ^
```

Custom Class Field Getters and Setters

Key points

- Kotlin classes can have *properties* (what we call *fields* in Java)
- Properties can be defined as `val` or `var`
- `val` fields are read-only, `var` fields are read-write
- Fields are accessed directly by their name, just like constructor parameters
- TODO: Properties are public by default, but can also be private
- You can write custom getter and setter (accessor and mutator) methods for your properties with a special syntax
- The getter/setter methods can use a “backing field” named `field` (TODO: `field` is kind of a reserved word in Kotlin but not really)

A field without custom getters or setters

If you just want a simple property in a class that you can read and modify, declare the field as a `var`:

```
class Person() {
    var name = ""
}

// create an instance
>>> val p = Person()

// no initial value
>>> p.name

// set the value
>>> p.name = "Al"

// get (access) the value
>>> p.name
Al
```

That shows how to get and set values of a simple property.

Custom get/set methods

Conversely, if you want to create custom get/set methods, define custom `get()` and `set()` methods just below the property. For example, imagine that you want to make a log entry every time someone accesses or updates the `name` of a `Person`:

```
class Person {
    var name: String = "<no name>"
        get() {
            println("OMG, someone accessed 'name'")
            return field
        }
        set(s) {
            println("OMG, someone updated 'name' to be '$s'")
            field = s
        }
}
```

The REPL shows how this works:

```
>>> val p = Person()

>>> p.name
OMG, someone accessed 'name'
<no name>

>>> p.name = "Al"
OMG, someone updated 'name' to be 'Al'

>>> p.name
OMG, someone accessed 'name'
Al
```

Key points:

- The getter/setter methods can use a "backing field" named `field`
- `field` is a way of referencing the value you're referencing, in this case the `name` field
- `field` is made available to you, but you don't have to use it
- Per the official Kotlin documentation, "A backing field will be generated for a property if it uses the default implementation of at least one of the accessors, or if a custom accessor references it through the field identifier."

More examples

Here are a few more examples of Kotlin's getter/setter approach.

First, a one-line getter with a multiline setter, both using the special backing field:

```
class Foo {
    var prop: String = "initial value"
        get() = field  //no real need for this, just showing syntax
        set(s) {
            // do whatever you want with `s` ...
            // then:
            field = s
        }
}
```

Second, multiline getters and setters:

```
class Bar {
    var name = ""
        get() {
            val s = if (field == "") "hello" else "hello, ${field}"
            return s
        }
        set(s) {
            // do whatever you want with `s`
            field = s
        }
}
```

This is how you can use your own "backing field" instead of `field`:

```
class Baz {
    private var _name = ""
    var name = ""
        fun get() = _name
        fun set(s: String) {
```

```
            _name = s
        }
}
```

Finally, here's an example that shows get/set methods alongside other functions in a class:

```
class BlogPost {
    var content = ""
        get() {
            logAccess(Date())
            return field
        }
        set(s) {
            logUpdate(Date(), s)
            field = s
        }
    fun logAccess(d: Date) {
        println("logAccess called ...")
        //...
    }
    fun logUpdate(d: Date, s: String) {
        println("logUpdate called ...")
        //...
    }
}
```

Constructor Default Values and Named Arguments

Kotlin has two nice features that you'll also find in Scala:

- You can supply default values for constructor parameters
- You can use named arguments when calling a constructor

Default values for constructor parameters

A convenient Kotlin feature is that you can supply default values for constructor parameters. For example, you *could* define a `Socket` class like this:

```
class Socket(var timeout: Int, var linger: Int) {
    override def toString = s"timeout: $timeout, linger: $linger"
}
```

That's nice, but you can make this class even better by supplying default values for the `timeout` and `linger` parameters:

```
class Socket(var timeout: Int = 2000, var linger: Int = 3000) {
    override fun toString(): String = "timeout: ${timeout}, linger: ${linger}"
}
```

By supplying default values for the parameters, you can now create a new `Socket` in a variety of different ways:

```
Socket()
Socket(1000)
Socket(4000, 6000)
```

This is what those examples look like in the REPL:

```
>>> Socket()
timeout: 2000, linger: 3000

>>> Socket(1000)
timeout: 1000, linger: 3000

>>> Socket(4000, 6000)
timeout: 4000, linger: 6000
```

As those examples shows:

- When all parameters have default values, you don't have to provide any values when creating a new instance
- If you supply one value, it's used for the first named parameter
- You can override the default values with your own values

An important implication of this is that default values have the effect of letting consumers consumers create instances of your class in a variety of ways — in a sense they work just as though you had created multiple, different constructors for your class.

When you don't provide defaults for all parameters

As a word of caution, it generally doesn't make any sense to provide a default value for an early parameter without providing a default for subsequent parameters.

```
// don't do this
class Socket(var timeout: Int = 5000, var linger: Int) {
    override fun toString(): String = "timeout: ${timeout}, linger: ${linger}"
}
```

If you do that, you'll get errors when trying to construct an instance with zero or one parameters:

```
>>> val s = Socket()
error: no value passed for parameter 'linger'
val s = Socket()
               ^

>>> val s = Socket(2)
error: no value passed for parameter 'linger'
val s = Socket(2)
                ^
```

If you're not going to provide default values for all parameters, you should only provide default values for the last parameters in the constructor:

```
// this is a little better
class Socket(var timeout: Int, var linger: Int = 5000) {
    override fun toString(): String = "timeout: ${timeout}, linger: ${linger}"
}
```

With this approach the zero-argument constructor still fails, but you can use the one-argument constructor:

```
>>> val s = Socket()
error: no value passed for parameter 'timeout'
val s = Socket()
               ^

>>> val s = Socket(10)
>>> s
timeout: 10, linger: 5000
```

Named arguments

You can use named arguments when creating new class instances. For example, given this class:

```
class Socket(var timeout: Int, var linger: Int) {
    override fun toString(): String = "timeout: ${timeout}, linger: ${linger}"
}
```

you can create a new `Socket` like this:

```
val s = Socket(timeout=2000, linger=3000)
```

I don't use this feature too often, but it comes in handy in certain situations, such as when all of the class constructor parameters have the same type (such as `Int` in this example). For example, some people find that this code:

```
val s = new Socket(timeout=2000, linger=3000)
```

is more readable than this code:

```
val s = new Socket(2000, 3000)
```

Secondary Class Constructors

Key points

Here are a few rules to know about Kotlin secondary class constructors:

- A class can have zero or more secondary class constructors
- A secondary constructor must call the primary constructor; this can happen by directly calling the primary constructor, or by calling another secondary constructor that calls the primary constructor
- You call other constructors of the same class with the `this` keyword
- The `@JvmOverloads` annotation lets Kotlin classes that have default parameter values be created in Java code

Secondary constructor examples

Here's an example that shows a primary constructor and two different auxiliary constructors:

```
class Pizza constructor (
    var crustSize: String,
    var crustType: String,
    val toppings: MutableList<String> = mutableListOf()
) {

    // secondary constructor (no-args)
    constructor() : this("SMALL", "THIN")

    // secondary constructor (2-args)
    constructor(crustSize: String, crustType: String) : this(crustSize, crustType, mutableListOf<String>())

    override fun toString(): String = "size: ${crustSize}, type: ${crustType}, toppings: ${toppings}"

}
```

This function and its output shows how the constructors work:

```
fun main(args: Array<String>) {
    val p1 = Pizza()
    val p2 = Pizza("LARGE", "THICK")
    val p3 = Pizza("MEDIUM", "REGULAR", mutableListOf("CHEESE", "PEPPERONI"))

    println(p1)
    println(p2)
    println(p3)
}

// output
size: SMALL, type: THIN, toppings: []
size: LARGE, type: THICK, toppings: []
size: MEDIUM, type: REGULAR, toppings: [CHEESE, PEPPERONI]
```

Notes

Notice how default constructor values eliminate the need for secondary constructors like these. This code with default constructor parameter values produces the same results:

```
class Pizza2 constructor (
    var crustSize: String = "SMALL",
    var crustType: String = "THIN",
    val toppings: MutableList<String> = mutableListOf()
) {
    override fun toString(): String = "size: ${crustSize}, type: ${crustType}, toppings: ${toppings}"
}
```

@JvmOverloads (Working with Java)

Per the Kotlin documentation, the `@JvmOverloads` annotation "Instructs the Kotlin compiler to generate overloads for this function that substitute default parameter values."

This doesn't do too much for you when you're working only in Kotlin, but it's necessary if you want to generate multiple constructors that will work with Java code. For example, this Kotlin code works just fine:

```
class Person constructor(
    var firstName: String,
    var lastName: String,
    var age: Int = 0
) {
    override fun toString(): String =
        "$firstName $lastName is $age years old"
}

fun main(args: Array<String>) {
    val p1 = Person("Alvin", "Alexander", 40)
    val p2 = Person("Alvin", "Alexander")

    println(p1)
    println(p2)
}
```

But if you try to use the `Person` class constructors in Java you'll find that the second line won't compile:

```
Person p1 = new Person("John", "Doe", 30);
Person p2 = new Person("Jane", "Doe");      //won't compile
```

The solution to work properly with Java is to add the `@JvmOverloads` annotation to the Kotlin class:

```
class Person @JvmOverloads constructor(  //same as before...
             -------------
```

With the rest of the class being the same, this annotation allows the Kotlin class to be used in Java code.

Open and Final Classes

Key points

- Because of a concern about the “fragile base class” problem, Kotlin classes and their functions are `final` by default
- To allow a class to be extended it must be marked `open`, meaning that it’s “open to be extended”
- To allow class functions and fields to be overridden, they must also be marked `open`

Classes and functions are final by default

I’ll demonstrate how this works through a series of examples.

(1) Class and function are not open

In this first example, `Child` won’t compile because `Parent` and its function `name` are `final` (by default):

```
class Parent {            //class is final by default
    fun name() = "base"   //function is final by default
}

// this won’t compile
class Child : Parent() {
    override fun name() = "Child"
}

// error messages:
error: this type is final, so it cannot be inherited from
class Child : Parent() {
              ^
error: 'name' in 'Parent' is final and cannot be overridden
    override fun name() = "Child"
    ^
```

(2) Class is open, function is not

In this example, `Parent` is now open, but because `name` isn’t open the code still won’t compile:

```
open class Parent {
    fun name() = "base"
}

class Child : Parent() {
    override fun name() = "Child"   //intentional error
}

// error message:
error: 'name' in 'Parent' is final and cannot be overridden
    override fun name() = "Child"   //intentional error
    ^
```

(3) Success: Class and function are open

Finally, with `Parent` and `name` both being marked `open`, this code will compile:

```
open class Parent {
    open fun name() = "base"
}

class Child : Parent() {
    override fun name() = "Child"
}
```

Also, note the need for the `override` modifier on `Child::name`.

Closing an open method

To begin looking at an interesting problem, notice that `foo` is defined to be `open` in class `A`, and then it is overridden in both class `B` and then class `C`:

```
open class A {
    open fun foo() = "foo in A"
}

open class B : A() {
    override fun foo() = "foo in B"
}

open class C : B() {
    override fun foo() = "foo in C"
}
```

This code is perfectly legal; so far, so good.

Now, if you're writing class `B` and don't want your subclasses to override `foo`, you can mark `foo` as `final` in your class:

```
open class A {
    open fun foo() = "foo in A"
}

open class B : A() {
    // added `final` here
    final override fun foo() = "foo in B"
}

open class C : B() {
    // this won't compile because `foo` is marked
    // `final` in B
    override fun foo() = "foo in C"
}

// error:
error: 'foo' in 'B' is final and cannot be overridden
```

Similarly, if you take the `open` off of class `B`, class `C` will no longer compile:

```
open class A {
    open fun foo() = "foo in A"
}

// removed `open` here
class B : A() {
    override fun foo() = "foo in B"
}
```

```
// this code won't compile because B is closed
open class C : B() {
    override fun foo() = "foo in C"
}

// error message:
error: this type is final, so it cannot be inherited from
open class C : B() {
               ^
```

More information: Fragile base classes

To understand Kotlin's approach with classes and functions, you first have to understand the "fragile base class" problem. Wikipedia describes it like this:

"The fragile base class problem is a fundamental architectural problem of object-oriented programming systems where base classes (superclasses) are considered 'fragile' because seemingly safe modifications to a base class, when inherited by the derived classes, may cause the derived classes to malfunction. The programmer cannot determine whether a base class change is safe simply by examining the methods of the base class in isolation."

This is a problem in Java, where classes and methods are open by default. To fix the problem, as Joshua Block writes in his classic book, Effective Java, to protect against this problem, developers should "Design and document for inheritance or else prohibit it."

Because of this experience in Java, the Kotlin designers decided to make all classes and methods `final` by default. That way, you can "design for inheritance" by marking classes and functions as `open` only when you really want them to be open to extension.

Abstract Classes

Abstract classes in Kotlin are similar to abstract classes in Java and Scala, but Kotlin makes a clear distinction about which methods are considered “open” to be overridden.

Key points

- Abstract classes are similar to Java’s abstract classes
- If you declare a class `abstract` it can’t be instantiated
- Abstract classes can have:
  - Abstract and concrete methods
  - Abstract and concrete properties (fields)
- Abstract classes and abstract members don’t have to be annotated with `open`; the fact that they are abstract implies that they are intended to be overridden
- Abstract classes can have private and protected members

Functions in abstract classes

Functions can be:

- Abstract: function signature only, no body
- Concrete, but closed (i.e., final)
- Concrete, and open to modification

Example:

```
abstract class Pet (name: String) {
    abstract fun comeToMaster(): Unit          //abstract, open by default
    fun walk(): Unit = println("I’m walking")  //concrete, closed (final)
    open fun speak(): Unit = println("Yo")     //concrete, open
}
```

Working code

Here’s a small, complete example that shows how abstract classes and functions work with inheritance:

```
abstract class Pet (name: String) {
    abstract fun comeToMaster(): Unit          //abstract, open by default
    fun walk(): Unit = println("I’m walking")  //concrete, closed (final)
    open fun speak(): Unit = println("Yo")     //concrete, open
}

class Dog(name: String) : Pet(name) {
    override fun speak() = println("Woof")
    override fun comeToMaster() = println("Here I come!")
}

class Cat(name: String) : Pet(name) {
    override fun speak() = println("Meow")
    override fun comeToMaster() = println("That’s not gonna happen")
}

fun main(args: Array<String>) {
```

```
    val d = Dog("Zeus")
    d.walk()
    d.speak()
    d.comeToMaster()

    val c = Cat("Rusty")
    c.walk()
    c.speak()
    c.comeToMaster()

}
```

Other abstract class features

Abstract classes can also have private properties, abstract properties, and concrete properties, as shown in the last three lines of this example:

```
abstract class Pet (name: String) {
    abstract fun comeToMaster(): Unit          //abstract method
    fun walk(): Unit = println("I'm walking")  //concrete, closed (final)
    open fun speak(): Unit = println("Yo")     //concrete, open

    private var numberOfLegs = 4       //private
    abstract var furColor: String      //abstract (no initial value)
    var actuallyLikesPeople = false    //concrete, must have a value
}
```

Why use abstract classes (instead of interfaces)?

Interfaces can't have constructor parameters, initialized properties, or private properties.

You can kind of simulate properties with default values, but interface properties can't have backing fields:

```
interface Cat {
    private var numLegs: Int
        get() = 4
        set(value) = TODO()   //can't use `field`
}
```

A note about designing base classes

This Reddit post has an interesting discussion about designing base classes. One interesting quote:

> "Designing a base class, you should therefore avoid using open members in the constructors, property initializers, and init blocks."

See that Reddit link and this Kotlin link for more details.

Kotlin Interfaces

If you're familiar with Java, Kotlin interfaces are similar to Java 8 interfaces. If you know Scala, they also have similarities to Scala traits.

Key points

What you'll see in this lesson:

- Interfaces are defined using the `interface` keyword
- Interfaces can declare functions, which can be abstract or concrete
- Interfaces can declare fields (properties):
  - They can be abstract
  - They can provide implementations for accessors
- Interfaces can inherit/derive from other interfaces
- A class or object can implement one or more interfaces
- When you inherit from multiple interfaces that have methods with the same names and signatures, you must manually resolve the conflict

As shown previously, Kotlin also has a concept of abstract classes. Use abstract classes instead of interfaces when:

- You need constructor parameters (interfaces can't have them)
- You need to define concrete read/write fields
- You need private members

Interfaces with abstract and concrete functions

This example shows common uses of interfaces, with both abstract and concrete functions in the interfaces, along with overridden functions in the concrete `Dog` class:

```
package interfaces

interface Speaker {
    //abstract function
    fun speak(): String
}

interface TailWagger {
    // concrete implementations
    fun startTail() { println("tail is wagging") }
    fun stopTail() { println("tail is stopped") }
}

class Dog : Speaker, TailWagger {

    // override an abstract function
    override fun speak(): String = "Woof!"

    // override a concrete function
    override fun stopTail() { println("can't stop the tail!") }

}

fun main(args: Array<String>) {
    val d = Dog()
    println(d.speak())  //"Woof!"
```

```
    d.startTail()       //"tail is wagging"
    d.stopTail()        //"can't stop the tail!"
}
```

Properties

Key points:

- Interfaces can define properties (fields)
- They can be abstract
- You can define an accessor (getter) for a property

Examples:

```
interface PizzaInterface {
    var numToppings: Int      //abstract
    var size: Int             //abstract
    val maxNumToppings: Int   //concrete
        get() = 10
}

class Pizza : PizzaInterface {

    // simple override
    override var numToppings = 0

    // override with get/set
    override var size: Int = 14  //14"
        get() = field
        set(value) { field = value}

    // override on a `val`
    override val maxNumToppings: Int
        //get() = super.maxNumToppings
        get() = 20
}


fun main(args: Array<String>) {
    val p = Pizza()
    println(p.numToppings)     //0
    println(p.size)            //14
    println(p.maxNumToppings)  //20

    p.numToppings = 5
    p.size = 16
}
```

A note from the official Kotlin docs: "A property declared in an interface can either be abstract, or it can provide implementations for accessors. Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them."

Inheritance

Interfaces can extend other interfaces, override their properties and functions, and declare new properties and functions:

```
interface StarTrekCrewMember {
    fun uniformColor(): String
}
```

```
interface Officer : StarTrekCrewMember {
    override fun uniformColor(): String = "not red"
}

interface RedShirt : StarTrekCrewMember {
    override fun uniformColor(): String = "red (sorry about your luck)"
    fun diePainfulDeath(): String = "i’m dead"
}

// more code ...
```

Here’s a class that implements three interfaces:

```
interface Starship
interface WarpCore
interface WarpCoreEjector
class Enterprise : Starship, WarpCore, WarpCoreEjector
```

Resolving inheritance conflicts

When a class extends multiple interfaces and those interfaces have a common method, you must manually handle the situation in the function in your class. This example shows how to handle the functions `foo` and `bar` in the class `C`, where `C` extends both interfaces `A` and `B`:

```
interface A {
    fun foo() { println("foo: A") }
    fun bar(): Unit
}

interface B {
    fun foo() { println("foo: B") }
    fun bar() { println("bar: B") }
}

class C : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
        println("foo: C")
    }
    override fun bar() {
        super<B>.bar()
        println("bar: C")
    }
}

fun main(args: Array<String>) {
    val c = C()
    c.foo()
    println()
    c.bar()
}
```

Here’s the output from `main`:

```
foo: A
foo: B
foo: C

bar: B
bar: C
```

> This example is a simplified version of this kotlinlang.org example.

Object Equality

Comparing objects — class instances — in Kotlin is a little different than Java, and very similar to Scala.

Key points

- `==` calls `equals` under the hood (structural equality)
- `===` is used to test reference equality
- Classes don't have `equals` or `hashCode` methods by default, you need to implement them
- *Data classes* have automatically generated `equals` and `hashCode` methods

`==` and `===`

This code shows how object equality works when a class *does not* have an `equals` method (`Person`) and when a class *does* have an `equals` method (`PersonWithEquals`):

```
class Person(var name: String)

class PersonWithEquals(var name: String) {
    override fun equals(that: Any?): Boolean {
        if (that == null) return false
        if (that !is PersonWithEquals) return false
        if (this.name == that.name) {
            return true
        } else {
            return false
        }
    }
}

fun main(args: Array<String>) {

    // Person (without `equals`)
    val p1 = Person("Joe")
    val p2 = Person("Joe")
    println(p1 == p2)   //false
    println(p1 === p2)  //false
    println(p1 === p1)  //true

    // PersonWithEquals
    val p1a = PersonWithEquals("Joe")
    val p2a = PersonWithEquals("Joe")
    println(p1a == p2a)   //true
    println(p1a === p2a)  //false
    println(p1a === p1a)  //true

}
```

Notes:

- You can write the `equals` function differently; I'm just trying to be obvious about the tests
- In the real world you should implement `hashCode`
- As you'll see in the Data Classes lesson, a *data class* implements `equals` and `hashCode` for you
- See the "nullability" lessons for the meaning of `Any?`

Summary

- Kotlin enumerations are similar to enumerations in other languages, such as Java
- Enumerations are declared with `enum class`
- Enumerations can have constructor parameters, properties, and functions
- Enumerations can be used in `if` and `when` expressions and `for` loops

Basic enumerations

Simple enumerations are a useful tool for creating small groups of constants, things like the days of the week, months in a year, suits in a deck of cards, etc., situations where you have a group of related, constant values. Here are the days of the week:

```
enum class DayOfWeek {
    SUNDAY, MONDAY, TUESDAY,
    WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

Here's an enumeration for the suits in a deck of cards:

```
enum class Suit {
    CLUBS, SPADES, DIAMONDS, HEARTS
}
```

The full power of enumerations

More than just simple constants, enumerations can have constructor parameters, properties, and functions, just like a regular class. Here's an example, adapted from this Oracle Java example:

```
enum class Planet(val mass: Double, val radius: Double) {
    // just the first three planets
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6);  //semi-colon is required

    // universal gravitational constant (m3 kg-1 s-2)
    val G = 6.67300E-11

    fun surfaceGravity(): Double {
        return G * mass / (radius * radius);
    }

    fun surfaceWeight(otherMass: Double): Double {
        return otherMass * surfaceGravity();
    }
}

fun main(args: Array<String>) {
    val earthWeight = 200.0
    val mass = earthWeight / Planet.EARTH.surfaceGravity()
    for (p in Planet.values()) {
        val s = String.format("Your weight on %s is %.2f", p, p.surfaceWeight(mass))
        println(s)
    }
}
```

Here's the output from that code:

```
Your weight on MERCURY is 75.55
Your weight on VENUS is 181.00
Your weight on EARTH is 200.00
```

Enumerations with `if`, `when`, and `for`

Enumerations can be used with `if` and `when` expressions and `for` loops, as shown in the following examples.

`if` expressions:

```
enum class DayOfWeek {
    SUNDAY, MONDAY, TUESDAY,
    WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

fun main(args: Array<String>) {
    val today = DayOfWeek.MONDAY

    if (today == DayOfWeek.MONDAY) {
        println("MONDAY")
    }
    else if (today == DayOfWeek.TUESDAY) {
        println("TUESDAY")
    }
}
```

`when` expressions:

```
enum class Margin {
    TOP, RIGHT, BOTTOM, LEFT
}

fun getMarginValue(margin: Margin) = when(margin) {
    Margin.TOP    ->  "1em"
    Margin.RIGHT  ->  "12px"
    Margin.BOTTOM ->  "1.5em"
    Margin.LEFT   ->  "6px"
}

fun main(args: Array<String>) {
    println(getMarginValue(Margin.TOP))
}
```

`for` loops:

```
for (m in Margin.values()) println(m)
```

A Complete Kotlin Class

In this lesson you'll create a `Pizza` class in Kotlin that might be used in a pizza-store point of sales system. The class will be written in an object-oriented programming (OOP) style, and will use enumerations, constructor parameters, and class properties and functions.

Pizza-related enumerations

I'll start creating a pizza class by first creating some enumerations that will be helpful:

```
enum class Topping {
    CHEESE, PEPPERONI, SAUSAGE, MUSHROOMS, ONIONS
}

enum class CrustSize {
    SMALL, MEDIUM, LARGE
}

enum class CrustType {
    REGULAR, THIN, THICK
}
```

Those enumerations provide a nice way to work with pizza toppings, crust sizes, and crust types.

A sample Pizza class

Given those enumerations I can define a `Pizza` class like this:

```
class Pizza (
    var crustSize: CrustSize = CrustSize.MEDIUM,
    var crustType: CrustType = CrustType.REGULAR
) {
    val toppings = mutableListOf<Topping>()

    fun addTopping(t: Topping): Boolean = toppings.add(t)
    fun removeTopping(t: Topping): Boolean = toppings.remove(t)
    fun removeAllToppings(): Unit = toppings.clear()
}
```

> `toppings` can be a constructor parameter, but I wrote it this way to show a class property.

When you save the enumerations and that class in a file named *Pizza.kt*, you can compile it with `kotlinc`:

```
$ kotlinc Pizza.kt
```

That command creates the following class files:

```
CrustSize.class
CrustType.class
Pizza.class
Topping.class
```

There's nothing to run yet because this class doesn't have a `main` method, but ...

A complete Pizza class with a main method

To see how this class works, replace the source code in *Pizza.kt* with the following code:

```
enum class Topping {
    CHEESE, PEPPERONI, SAUSAGE, MUSHROOMS, ONIONS
}

enum class CrustSize {
    SMALL, MEDIUM, LARGE
}

enum class CrustType {
    REGULAR, THIN, THICK
}

class Pizza (
    var crustSize: CrustSize = CrustSize.MEDIUM,
    var crustType: CrustType = CrustType.REGULAR
) {

    val toppings = mutableListOf<Topping>()

    fun addTopping(t: Topping): Boolean = toppings.add(t)
    fun removeTopping(t: Topping): Boolean = toppings.remove(t)
    fun removeAllToppings(): Unit = toppings.clear()

    override fun toString(): String {
        return """
        |Pizza:
        |  Crust Size: $crustSize
        |  Crust Type: $crustType
        |  Toppings:   $toppings
        """.trimMargin()
    }
}

fun main(args: Array<String>) {
    val p = Pizza()
    p.addTopping(Topping.CHEESE)
    p.addTopping(Topping.PEPPERONI)
    p.addTopping(Topping.MUSHROOMS)
    p.addTopping(Topping.ONIONS)
    println(p)
    p.removeTopping(Topping.MUSHROOMS)
    p.removeTopping(Topping.ONIONS)
    println(p)
    p.removeAllToppings()
    println(p)
}
```

Notice that I added a `toString` function to the `Pizza` class, and added a `main` function to test everything in the `Pizza` class.

When you compile the code with this command:

```
$ kotlinc Pizza.kt -include-runtime -d Pizza.jar
```

then execute the jar file with this `java` command:

```
$ java -jar Pizza.jar
```

you'll see this output:

```
Pizza:
  Crust Size: MEDIUM
  Crust Type: REGULAR
  Toppings:   [CHEESE, PEPPERONI, MUSHROOMS, ONIONS]
Pizza:
  Crust Size: MEDIUM
  Crust Type: REGULAR
  Toppings:   [CHEESE, PEPPERONI]
Pizza:
  Crust Size: MEDIUM
  Crust Type: REGULAR
  Toppings:   []
```

This is a good start for creating an OOP-style class that uses enumerations, constructor parameters, and class properties and functions.

Data Classes

Sometimes you just need to create classes to hold data, like structs in the C programming language. Kotlin *data classes* give you a way to create data structures that have automatically-generated functions that make them very useful. If you're familiar with Scala's *case classes*, data classes are almost exactly the same.

> Data classes are generally intended for use in a functional programming (FP) environment, but OOP developers can use them as well.

Key points

- In FP you create classes just to hold data, like structs in C
- Kotlin *data classes* are useful for this purpose
- The compiler automatically generates the following functions for data classes:
  - `equals()` and `hashCode()` methods
  - a useful `toString()` method
  - a `copy()` function that is useful in an update as you copy scenario
  - "componentN" functions are created to let you destructure an instance into its constructor fields
- Data classes are like regular Kotlin classes, but because they're intended to hold data, the primary constructor must have at least one parameter

Functional programming

While data classes may be useful for OOP, they're primarily intended for the FP paradigm:

- FP developers create data classes with all `val` constructor parameters
- All instances of data classes are also `val`

In FP, all variables and all collections are immutable, so the features of data classes are especially useful when everything is immutable, the only way to update variables is to update them as you make a copy of them.

> See my book, Functional Programming, Simplified for *many* more details on this.

Example

First, here are some operations on a regular Kotlin class:

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

fun main(args: Array<String>) {

    val p1 = Person("Jane Doe", 30)
    val p2 = Person("Jane Doe", 30)

    println(p1.name)   //Jane Doe
    println(p1.age)    //30

    println(p1 == p2)  //false
    println(p1)        //Person@49476842
```

```
}
```

Notice that `p1` does not equal `p2` and `p1` does not print very well.

Next, just put the keyword `data` before the class to create a data class:

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

fun main(args: Array<String>) {

    val p1 = Person("Jane Doe", 30)
    val p2 = Person("Jane Doe", 30)

    println(p1.name)   //Jane Doe
    println(p1.age)    //30

    println(p1 == p2)  //true
    println(p1)        //Person(name=Jane Doe, age=30)

    val p3 = p2.copy(name = "Jane Deer")
    println(p3)        //Person(name=Jane Deer, age=30)

}
```

Notice:

- `p1` equals `p2` (because `equals` and `hashCode` are generated for you)
- `Person` is readable when you print it out
- Use the `copy` function to create a modified variable

Getter/setter functions for constructor parameters

For OOP developers, data class constructor parameters can also be a `var`:

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

fun main(args: Array<String>) {
    val p = Person("Jane Doe", 30)
    p.name = "Jane Deer"  //setter function
    p.age = 31            //setter function
    println(p)            //Person(name=Jane Deer, age=31)
}
```

Key points:

- `var` constructor parameters will have both getter and setter methods
- `val` constructor parameters will have only getter methods
- All constructor parameters are included in the `toString()` output

Data class properties don’t print

Data classes can also have properties — such as `age` in this next example — but notice that they don't print out; only constructor fields are included in `toString()` output:

```
data class Person (var name: String) {
    var age: Int = 0
}

fun main(args: Array<String>) {
    val p = Person("Jane Doe")
    p.age = 30
    println(p)   //Person(name=Jane Doe)  <-- no age here
}
```

`equals()` and `hashCode()`

As shown earlier, data classes have automatically-generated `equals` and `hashCode` methods, so instances can be compared:

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

val p1 = Person("Jane Doe", 30)
val p2 = Person("Jane Doe", 30)

println(p1 == p2)  //true
```

These methods also let you easily use data class instances in collections like sets and maps.

`toString()`

As shown, data classes have a good default `toString()` method implementation, which at the very least is helpful when debugging code:

```
val p = Person("Jane Doe", 30)
p   //Person(name=Jane Doe, age=30)
```

"componentN" functions

Functions are generated for data classes that enable them to be used in destructuring declarations:

```
// create a Person
val p = Person("Jane Doe", 30)

// destructure the person into its fields
val (name, age) = p
println("$name is $age")   //"Jane Doe is 30"

// field names aren't important
val (a, b) = p
println("$a is $b")        //"Jane Doe is 30"
```

Only constructor fields are destructured this way.

Copying

As shown earlier, a data class also has an automatically-generated `copy` method that's extremely helpful when you need to perform the process of a) cloning an object and b) updating one or more of the fields during the cloning process:

```
> data class BaseballTeam(val name: String, val lastWorldSeriesWin: Int)
> val cubs1908 = BaseballTeam("Chicago Cubs", 1908)
> val cubs2016 = cubs1908.copy(lastWorldSeriesWin = 2016)

> cubs1908
BaseballTeam(name=Chicago Cubs, lastWorldSeriesWin=1908)

> cubs2016
BaseballTeam(name=Chicago Cubs, lastWorldSeriesWin=2016)
```

When you use the `copy` method just supply the names of the fields you want to modify during the cloning process.

Because you never mutate data structures in FP, this is how you create a new instance of a class from an existing instance. I refer to this process as "update as you copy," and I discuss this process in detail in my book, Functional Programming, Simplified.

Create Singletons with Object

Key points

- Use `object` instead of `class` to create a singleton (single instance of that class)
- Useful for “utility” classes

Rules:

- No constructor parameters
- Can contain:
  - Properties
  - Functions
  - Initializer blocks
- Objects can inherit from interfaces and classes
- There is a difference between object *expressions* and *declarations*
  - Expressions are used on the right hand side of expressions
  - Declarations are when you use `object` to create a singleton or companion object

Singleton example

```
object MathUtils {
    init {
        println("MathUtils init was called")
    }
    val ONE = 1
    fun plus1(i: Int) = i + 1
}

fun main(args: Array<String>) {
    val x = MathUtils.plus1(MathUtils.ONE)
    println(x)
}
```

Prints:

```
MathUtils init was called
2
```

Inheritance

This is from the Kotlin Language Reference (TODO: write my own example):

```
// objects can have supertypes:
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }
    override fun mouseEntered(e: MouseEvent) { ... }
}
```

Companion Objects (TODO)

- Similar to Scala
- Create a companion object inside a class
- A way of adding static methods to a class
- Factory pattern: private constructor in class + factory method in companion object
- TODO: Document rules about relationship between a class and its companion object

Syntax

From Kotlin Language Reference (TODO: write my own example):

(1) Give the companion object a name:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

val instance = MyClass.create()
```

(2) No name on companion object:

```
class MyClass {
    companion object { } // will be called "Companion"
}
fun MyClass.Companion.foo() { ... }
```

Notes

- Style/Idiom: Use package-level functions instead of static methods in companion objects? (Noted in Kotlin Language Reference)

TODO

- `@JvmStatic` annotation
- From Kotlin Language Reference, p. 83: “Note that, even though the members of companion objects look like static members in other languages, at runtime those are still instance members of real objects, and can, for example, implement interfaces:”

- public, private, etc.
- i think this goes here, maybe not
- TODO: need to discuss "module" (KIA, p.73)

Access modifiers in classes

From KIA, p.72

Modifier	Meaning	Comments
abstract	must be overridden	TODO
final	can’t be overridden	TODO
open	can be overridden	TODO
override	override a member in a superclass or interface	

Kotlin visibility modifiers

From KIA, p.73

Modifier	Class Member
public	visible everywhere
internal	visible in a module
protected	visible in subclasses
private	visible in a class

A First Look at Kotlin Functions

In Kotlin, *functions* can be defined both inside and outside of classes.

Key points

- Functions are defined with the `fun` keyword
- The syntax:

```
// single-expression syntax
fun plus1(i: Int) = i + 1
fun plus1(i: Int): Int = i + 1

// multiline syntax
fun plus1(i: Int): Int {
    return i + 1
}

// calling a function
val x = plus1(1)
```

- Functions are called just like you call Java methods
- Function parameters can have default values
- Functions can be called with named parameters
- Local functions can be created (functions defined within functions)
- “Infix” functions can be created (and will be covered in the next lesson)
- *Extension functions* let you extend existing types
- TODO: functions don't have to be inside classes, they can be at a package level

One-line functions (single-expression)

One-line functions are called “single-expression” functions and they can be written like this:

```
// single-expression syntax
fun plus1(i: Int) = i + 1
fun plus1(i: Int): Int = i + 1
```

Notice that the `return` keyword is not used with the single-expression syntax.

Multiline functions with return values

A multiline function looks like this:

```
fun addThenDouble(a: Int, b: Int): Int {
    val sum = a + b
    val doubled = sum * 2
    return doubled
}
```

The REPL shows how `addThenDouble` works:

```
> addThenDouble(1,1)
4

> addThenDouble(1,2)
6
```

Multiline functions without return values

If a multiline function doesn't have a return value, you can either leave it off the function declaration:

```
fun addAndPrint(a: Int, b: Int) {
    val sum = a + b
    println(sum)
}
```

or declare the return type as `Unit`:

```
fun addAndPrint(a: Int, b: Int): Unit {
    val sum = a + b
    println(sum)
}
```

Kotlin's `Unit` return type is similar to void/Void in other programming languages.

Function parameters can have default values

Just like constructors, you can specify default values for function parameters:

```
fun connect(timeout: Int = 5000, protocol: String = "http"): Unit {
    println("timeout = ${timeout}, protocol = ${protocol}")
    // more code here
}
```

Because of the default parameters, that function can be called in these ways:

```
connect()
connect(2500)
connect(10000, "https")
```

Here's what those examples look like in the REPL:

```
> connect()
timeout = 5000, protocol = http

> connect(2500)
timeout = 2500, protocol = http

> connect(10000, "https")
timeout = 10000, protocol = https
```

Functions can use named arguments

Just as with constructors, you can use named arguments when calling a function:

```
fun printName(firstName: String, lastName: String) {
    println("Your name is ${firstName} ${lastName}")
}
```

```
printName(firstName="Hala", lastName="Terra")
```

The REPL shows the result:

```
> printName(firstName="Hala", lastName="Terra")
Your name is Hala Terra
```

Local functions

When functions are only accessed by other functions in the local scope, you can declare those other functions to be `private`, but in Kotlin you can make the other functions local to the outer function. In this example the functions `add` and `double` are defined inside the outer `addThenDouble` function:

```
fun addThenDouble(a: Int, b: Int): Int {
    fun add(a: Int, b: Int) = a + b  //local function
    fun double(a: Int) = a * 2       //local function
    return double(add(a, b))
}
```

The REPL shows that this works as desired:

```
> addThenDouble(1,1)
4

> addThenDouble(1,2)
6
```

Benefits:

- Minimize the scope of functions
- Make it easier to understand the outer function because you don’t need to look around through other code to see the source code for the other functions that it uses

Extension Functions

Extension functions let you add new methods to existing classes (like `String`, `Int`, etc.). If you've used *implicit classes* in Scala, extension functions are just like those.

Example

In the years before Kotlin, I wanted an Android method that would let me write code like this:

```
if (x.between(1, 10)) ...
```

I couldn't do that with Java, so I had to settle for less-readable code that looked like this:

```
if (between(x, 1, 10)) { ...
```

But these days with Kotlin I can get what I want with an *extension function*:

```
fun Int.between(a: Int, b: Int): Boolean {
    return if (this >= a && this <= b) true else false
}
```

The REPL shows how to use that function:

```
>>> 5.between(1, 10)
true

>>> 22.between(1, 10)
false
```

How to declare an extension function

The key to writing an extension function is that you declare that you want to add a function to a specific data type:

```
fun Int.between(a: Int, b: Int): Boolean {
    ---
```

You also specify that you want the function to have this name:

```
fun Int.between(a: Int, b: Int): Boolean {
        -------
```

these parameters:

```
fun Int.between(a: Int, b: Int): Boolean {
                ------  ------
```

and this return type:

```
fun Int.between(a: Int, b: Int): Boolean {
```

```
                                -------
```

In fact, an extension function is just like a normal function, except that it's preceded by the name of the data type you want the function to be added to, as shown in this example.

An extension function on `Any`

Here's an extension function that I add to the `Any` type:

```
fun Any.printMe(): Unit = println("I am ${this}")
```

Because `Any` is the root of all Kotlin objects — like `java.lang.Object` in Java — it works on all Kotlin types:

```
1.printMe()
"Hello".printMe()
true.printMe()
```

Here's what those examples look like in the REPL:

```
>>> 1.printMe()
I am 1

>>> "Hello".printMe()
I am Hello

>>> true.printMe()
I am true
```

Infix Functions

An *infix function* is a function that appears in between two variables. Kotlin makes it easy to create functions like this.

Key points

- Infix functions are declared similarly to extension functions
- They are declared by writing `infix fun` at the start of the function definition
- When applied like `1 plus 2`, the function belongs to the variable on the left (`1`) and the variable on the right (`2`) is the function parameter
- Can be used to create DSLs

Examples

Imagine that the `+` operator doesn't exist and you want a function to add two integers, like this:

```
val x = 1 plus 1
```

Just write an infix function like this:

```
infix fun Int.plus(that: Int) = this + that
```

Here's how it looks in the Kotlin REPL:

```
> 1 plus 1
2
```

Notes

Note the `infix` keyword:

```
infix fun Int.plus(that: Int) = this + that
-----
```

Notice that it's an extension function for the `Int` type:

```
infix fun Int.plus(that: Int) = this + that
          ---
```

Understanding the "target" object

By changing the function:

```
infix fun Int.plus(that: Int): Int {
    println("this: ${this}, that: ${that}")
    return this + that
}
```

```
> 1 plus 2
this: 1, that: 2
3
```

You can see:

- The variable on the left (`1`) is considered the *target* object
- The variable on the right (`2`) is the function parameter

vararg Parameters

A Kotlin function parameter can be defined as a `vararg` parameter so it can be called with zero or more values.

Key points

- Declare function parameters with the `vararg` keyword
- When arrays are passed in, handle them with the `*` character
- Note: “In Kotlin, a vararg parameter of type `T` is internally represented as an array of type `T` (`Array<T>`) inside the function body.”

Basic example

Here’s a function that takes a `vararg` parameter:

```
fun printAll(vararg ints: Int) {
    for (i in ints) println(i)
}
```

The REPL shows how it works:

```
> printAll(1)
1

> printAll(1,2,3)
1
2
3

> printAll()
```

Passing in an array

Passing in an array won’t work:

```
val arr = intArrayOf(1,2,3)

> printAll(arr)
error: type mismatch: inferred type is Array<Int> but Int was expected
printAll(arr)
         ^
```

The solution is that you need to pass the array into your `vararg` parameter with this syntax:

```
printAll(*arr)
```

The `*` character in this use is known as the “spread operator.” The spread operator lets you convert arrays — well, most arrays — so they can be passed into a vararg parameter.

Not all arrays work (TODO)

The spread operator only works with certain types of arrays with vararg parameters. For example, I just showed that `intArrayOf` works:

```
> val intArray = intArrayOf(1,2,3)

> printAll(*intArray)
1
2
3
```

However, `arrayOf` does not work when it's given a list of integers:

```
> val arrayOfInt = arrayOf(1,2,3)

> printAll(*arrayOfInt)
error: type mismatch: inferred type is Array<Int> but IntArray was expected
printAll(*arrayOfInt)
          ^
```

To understand this it helps to look at the data types in the Kotlin REPL. Here are some details about the result from `intArrayOf`:

```
> val x = intArrayOf(1,2,3)

> println(x.javaClass.name)
[I

> println(x.javaClass.kotlin)
class kotlin.IntArray

> println(x.javaClass.kotlin.qualifiedName)
kotlin.IntArray
```

Similarly, here are some results for `arrayOf`:

```
> val x = arrayOf(1,2,3)

> println(x.javaClass.name)
[Ljava.lang.Integer;

> println(x.javaClass.kotlin)
class kotlin.Array

> println(x.javaClass.kotlin.qualifiedName)
kotlin.Array
```

As shown by those results, `intArrayOf` creates a type of `kotlin.IntArray`, which works with a vararg parameter of type `Int`, but `arrayOf` creates a type of `kotlin.Array`, which does not work with the same vararg parameter. (This is true as of Kotlin version 1.2.51.)

Bonus: Make it generic (TODO: NOT WORKING)

```
fun <T> printAll(vararg elems: T) {
    for (e in elems) println(e)
}
```

fun Int.toString() = this printAll(arrayOf(1,2,3)) printAll(intArrayOf(1,2,3))

val x = arrayOf(1,2,3) val x = intArrayOf(1,2,3)

Nullability: Introduction

Nullability is a Kotlin feature that's intended to help you eliminate *NullPointerException*s in your code. We all know that null pointer exceptions are bad, and the features I introduce in the following lessons are Kotlin's way of eliminating them.

Two rules about using null values

Before you move into the following lessons, it's worth mentioning that I like to think that there are two rules about using null values in Kotlin:

1. Never use null values!
2. When you're forced to use null values — such as by using an old Java library — use Kotlin's nullable types and the operators shown in the next few lessons.

With that point made, let's jump into nullable types.

Nullable Types

Nullable types are an approach in Kotlin to help you eliminate null values and null pointer exceptions (the dreaded `NullPointerException`). Here are the main concepts.

Key points

These are the key points of this lesson:

- A variable of type `String` can never be null
- A variable of the nullable type `String?` can be null
- The operations that can be performed on nullable types are very limited

I use `String` in those statements, but the same things can be said about every other type, including `Int`, `Float`, `Person`, etc.

Standard types can’t be null

Variables that are instances of standard Kotlin types can’t contain null values:

```
val> val s: String = null
error: null can not be a value of a non-null type String
val s: String = null
                ^

> val i: Int = null
error: null can not be a value of a non-null type Int
val i : Int = null
              ^
```

Similarly, variables that are instances of custom types can’t be null either:

```
> class Person (var name: String)

> val p: Person = null
error: null can not be a value of a non-null type Line_3.Person
val p: Person = null
                ^
```

Nullable types

A *nullable type* is a variation of a type that permits null values. You declare a type to be nullable by adding a question mark after it:

```
var s: String? = null
            _
```

This simple addition to your type allows your variable to contain null values. Notice that no exceptions are thrown in the REPL when you declare a nullable type:

```
> var s: String? = null
```

```
> var i: Int? = null
> var p: Person? = null
```

Also notice that I intentionally declare those variables as `var` fields. I do this because they're declared with null values, but at some point later in their lifetime they'll presumably contain more useful values.

Summary

A brief summary so far:

- Default Kotlin types cannot contain null values.
- A nullable type is a variation of an existing type, and can contain a null value. The `?` operator says, "This instance of this type is allowed to contain a null value."

As mentioned in the previous chapter, you should write your code to never use null values. Allowing null values in your code is considered a "bad smell." (When I look at code that permits null values I think, "How quaint, this is like code from 1996.")

Places where you'll use nullable types

You'll use and encounter nullable types in several areas:

- Variable assignment
- Function parameters
- Function return values
- Converting a nullable type to its non-nullable equivalent

Here's what those situations look like. You've already seen variable assignment:

```
var s: String? = null
```

This is a function parameter:

```
fun foo(s: String?) ...
```

This is a nullable type used as a function return value:

```
fun foo(): String? = ...
```

You may also need to handle the process of converting a nullable type to its non-nullable equivalent:

```
val x: String? = null
val y: String = x    // error: this won't compile
```

If you attempt to write the code as shown, you'll see this error message on the second line:

```
//error: type mismatch: inferred type is String? but String was expected
```

I'll show how to handle this situation in the following lessons.

Nullable types are limited

Because a nullable type can be `null`, in order for them to be safe to work with, the methods you can call on them are intentionally very restricted. As you can imagine, if you try to call a function like `length()` on a null string, it will throw a null pointer exception. Therefore, a `String?` instance doesn't have a `length` property:

```
> var s: String? = "fred"

> s.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed
on a nullable receiver of type String?
s.length
 ^
```

A `String?` instance also doesn't have a `toUpperCase()` function:

```
> s.toUpperCase()
error: only safe (?.) or non-null asserted (!!.) calls are allowed
on a nullable receiver of type String?
s.toUpperCase()
 ^
```

That error message is Kotlin's way of saying, "`String?` doesn't have this functionality."

This is an important point: a `String?` isn't the same as a `String`. For example, with Kotlin 1.5.2, a `String` instance has well over 100 functions that can be called on it, while a `String?` has only 13. You can think of `String?` as being a limited subset of a `String`.

> Note 1: Nullability concepts are enforced by the compiler at compile-time.
>
> Note 2: I'll discuss the `?.` and `!!.` operators you see referenced in those error messages in the lessons that follow.

Coming soon: Operators to help you

While you can work with nullable types manually by checking for null values in if/then expressions:

```
val len = if (s != null) s.length else 0
```

that code is verbose, and it only gets worse when you have several nullable types. Therefore, this style of code is considered a bad practice, and Kotlin has operators to help you work with nullable types. Those operators will be demonstrated in the lessons that follow.

The Safe-Call Operator

A main operator you'll use with nullable types is called the *safe-call operator*, and as you'll see in this lesson, it's represented by the symbol `?.` (a question mark followed by a period).

Background

If Kotlin had nullable types but didn't have any other operators to work with them, you'd always have to use if/then checks to work with them, like this:

```
fun toUpper(s: String?): String? = if (s != null) s.toUpperCase() else null
```

That function isn't too helpful, but it does help keep you from throwing null pointer exceptions. The REPL shows how it works:

```
>>> toUpper("al")
AL

>>> toUpper(null)
null
```

The safe-call operator

Fortunately Kotlin has a safe-call operator that does the same thing:

```
fun toUpper(s: String?): String? = s?.toUpperCase()
```

The REPL shows that this function works just like the previous one:

```
>>> toUpper("al")
AL

>>> toUpper(null)
null
```

In summary, these two pieces of code are equivalent:

Safe-Call Operator	Manual check
`s?.toUpperCase()`	`if (s != null) s.toUpperCase() else null`

Chaining safe calls together

In the real world, a field of one class may be a nullable type, and that class may have another nullable type, and so on. For example, in the following code `Order` has the nullable field `customer`; the `Customer` type has a nullable field `address`; and `Address` has a nullable field `street2`:

```
class Order (
    var customer: Customer?,
    var pizzas: List<Pizza>
```

```
)

class Customer (
    var name: String,
    var address: Address?
)

class Address (
    var street1: String,
    var street2: String?,
    var city: String,
    var state: String,
    var zip: String
)

class Pizza(
        //...
)
```

A great thing about the safe-call operator is that it can be chained together so you can access the `street2` field like this:

```
val street2Address = order.customer?.address?.street2
```

That syntax can be read as, “Reach inside the `order` instance and if the `customer` field is not null, and its `address` field is not null, give me the `street2` value (which may be null). However, if `customer` or `address` are null, give me a `null` value.”

It’s important to know that `street2Address` may still be null, but the key here is that this code won’t throw a `NullPointerException` if `customer`, `address`, or `street2` is null.

Complete code example

Here’s a larger example of using the safe-call operator:

```
class Order (
    var customer: Customer?,
    var pizzas: List<Pizza>
)

class Customer (
    var name: String,
    var address: Address?
)

class Address (
    var street1: String,
    var street2: String?,
    var city: String,
    var state: String,
    var zip: String
)

class Pizza(
    //...
)

fun main(args: Array<String>) {
    val address = Address(
        "123 Main",
        null,
        "Talkeetna",
```

```
        "Alaska",
        "99676"
    )

    // customer has no address
    var customer = Customer("Al", null)
    val order = Order(customer, emptyList<Pizza>())
    println("test1: ${order.customer?.address?.street2}")   //null

    // customer has address but no 'street2'
    customer.address = address
    println("test2: ${order.customer?.address?.street2}")   //null

}
```

Null object pattern

Depending on your needs, don't forget about the *Null Object Pattern*. Here's a link to a Scala null object pattern example I put together.

The Elvis Operator `?:`

The Safe-Call operator shown in the previous lesson is nice in many situations where a variable may contain a null value, and you're okay with ending up with another null value when you use it in an expression. But what about those times where you want to handle that null value and return something besides `null`? That's where the so-called "Elvis" Operator comes in.

> I don't see it myself, but I've read that if you turn your head sideways the `?:` characters look a little like Elvis. The official name for this operator is the "Null Coalescing Operator," and I think "Elvis Operator" is just easier to remember.

Key points

- `?:` comes after a nullable type instance and can be read as "or else" or "if that's null do this"
- `?.` returns null, but `?:` lets you return some other value besides `null`

Examples

In the previous lesson I returned `String?` from my `toUpper` function. I do that because the function accepts a nullable string, and may therefore return a null value:

```
fun toUpper(s: String?): String? = s?.toUpperCase()
                        ---------
```

The Elvis operator — `?:` — is nice because it lets you easily return something besides `null`:

```
fun toUpper(s: String?): String = s?.toUpperCase() ?: "DUDE!"
                        --------                   --
```

Here's how that function works:

```
>>> toUpper("foo")
FOO

>>> toUpper(null)
DUDE!
```

The use of the Safe-Call and Elvis operators in this example is equivalent to this code:

```
fun toUpper(s: String?): String {
    if (s != null) {
        return s.toUpperCase()
    } else {
        return "DUDE!"
    }
}
```

Discussion

The `Option` class in the Scala programming language has a method called `getOrElse`, and the `Optional` class in Java has an `orElse` method. I find that the Elvis operator reminds me of these methods. It's basically a way of saying "or else," which you can see in this example:

```
fun toUpper(s: String?): String = s?.toUpperCase() ?: "DUDE!"
                                  ----------------    ----------------
                                  if s is not null    "or else" return
                                     return this            this
```

You can also think of the Elvis operator as being an operator named `ifThatsNullReturnThis`:

```
fun toUpper(s: String?): String = s?.toUpperCase() ifThatsNullReturnThis: "DUDE!"
```

Another example

Here's another example you can use for testing:

```
class Person (
    var firstName: String,
    var mi: String?,
    var lastName: String
)

fun main(args: Array<String>) {
    var p: Person? = null
    p = Person("Alvin", null, "Alexander")
    val mi = p?.mi ?: "<blank>"
    println("mi = $mi")  //prints "mi = <blank>"
}
```

The `let` Operator

The `let` operator is an interesting construct that lets you run an algorithm on a variable inside a closure. When it's combined with the Safe-Call operator you can think of the approach as, "If the value exists, run this algorithm with the value." This is best shown by example.

`let` by itself

To get started, let's see how `let` works by itself — without the Safe-Call operator. Here's an example:

```
> val name = "foo"

> name.let { println("`it` is $it") }
`it` is foo
kotlin.Unit
```

TODO: IS "closure" THE CORRECT NAME FOR THIS?

The first line of the REPL output shows that the `println` statement is executed, and the value of `it` is the string `"foo"`. This is how `let` works; it makes the value of the variable it's applied to available inside the closure. The value in the closure is available through the variable `it`, but you can also use the "named parameter" approach, if you prefer:

```
// named parameter
name.let { s -> println("`s` is $s") }
           -                    --
```

That code works just like the previous `it` example.

`let` has a return value

Before you move on, notice that the second line of output in the Kotlin REPL is this:

```
kotlin.Unit
```

This line is printed because `let` has a return value. The previous example just printed to STDOUT, so there is no return value, so the REPL shows `kotlin.Unit`. Here's an example that has a more useful return value:

```
> "foo".let { it.length }
3
```

In this example `it` contains the string `"foo"`, and `"foo".length` is `3`, which is the value returned. In the real world you'll often assign the result from `let` to a variable, like this:

```
> val len = "foo".let { it.length }

> len
3
```

Using `let` with the Safe-Call operator

Now that you've seen how to use `let` by itself, let's look at how to use it in combination with the safe-call operator.

Given a `String?` value which may or may not be null:

```
val name: String? = null
//name might be changed later ...
```

You can print the value, if it exists:

```
name?.let { println("The name is $it") }
```

If `name` is null you won't see any output; `let` is smart enough to know that the value is null, so the closure won't be executed. But if it happens to contain a value, such as the string `"Al"`, you'll see output like this:

```
"The name is Al"
```

The advantage of `let`

Using `let` with the Safe-Call operator is the same as writing this code:

```
if (name != null) {
    // do something with `name`
}
```

The advantage of `let` is that it's more concise than constantly writing `if` statements like that.

A space before `let`

Note that if you prefer, you can also write that expression like this, with space before `let`:

```
name?. let { println("The name is $it") }
```

Some people find that approach more readable.

Lambda syntax and `let`

In a related note, you can write a lambda expression with `let` using a named parameter, like this:

```
// named param
toUpper(name)?.let { s -> println("Hello $s") }
                     -                   --
```

You can also write the lambda expression as I've shown previous in this lesson, with the special `it` variable name:

```
// `it`
toUpper(name)?.let { println("Hello $it") }
                                    ---
```

Key points

- if the value is null, `let` does nothing

- if the value is not null, `let` runs your algorithm
- `let` provides its own function scope and can be called on any value (NerdGuide)

TODO

- difference(s) between `let`, `run`, and `apply`
  - from my tests, `let` seems to create access to `it`, `run` does not
  - per book, "let returns the value of the closure"
    - per my test, `apply` doesn't create an `it` reference
- how `let` works:

```
"foo".let { println(it) } foo
```

The “Force” Operator (`!!`)

Kotlin also includes an operator generally known as the Force Operator. It lets you convert a nullable type to its non-nullable equivalent, such as converting a `String?` to a `String`, and it should only be used when you are 100% certain that a nullable type isn’t null.

> The official name of this operator is the “Non-null assertion operator.”

Examples

If you’re 100% certain that a nullable type doesn’t contain a null value, you can convert it to its non-nullable equivalent, like this:

```
> val s1: String? = "Christina"

> val s2: String = s1!!

> s2
Christina
```

However, beware that if the nullable type does contain a null value you’ll get a null pointer exception during this process:

```
> val s1: String? = null

> val s2: String = s1!!
kotlin.KotlinNullPointerException
```

Check for null before using `!!`

Getting a null pointer exception defeats the entire purpose of using nullable types, so the typical way of using the `!!` operator is to test that the nullable variable isn’t null before doing the assignment, like this:

```
var nullableName: String? = null

// nullableName may be reassigned ...

var name = ""

if (nullableName != null) {
    name = nullableName!!
}
```

Key points

A few key points about the Force operator:

- `!!` gives you a way to convert a nullable type to its non-nullable equivalent
- Be careful, it will throw a `kotlin.KotlinNullPointerException` if the value is null

And one note about when you should use the Force operator:

- Rarely!!

In general, there are better ways to use nullable types rather than converting them all to their non-nullable equivalent.

A Nullability Example

TODO: Show one or more examples of all of the operators working together.

Nullability Summary

Examples of working with nullable types:

```
> val s: String? = "Rocky"   //nullable type

> s?.length                  //safe-call operator
5

> s?.length ?: 0             //elvis operator
5

> s!!.length                 //force operator
5
```

More:

```
// Nullable type
var s: String? = null


// Safe-call operator
val street2Address = order.customer?.address?.street2


// Elvis operator (1)
fun toUpper(s: String?): String = s?.toUpperCase() ?: "DUDE!"

// Elvis operator (2)
var p: Person? = null
p = Person("Alvin", null, "Alexander")
val mi = p?.mi ?: "<blank>"
println("mi = $mi")


// Force operator (1)
> val s1: String? = "Christina"
> val s2: String = s1!!
> s2
Christina

// Force operator (2)
> val s1: String? = null
> val s2: String = s1!!
kotlin.KotlinNullPointerException
```

Nullability and collections (TODO)

```
listOfNotNull("a", "b", null)  //[a, b]
listOf("a", "b", null)         //[a, b, null]
```

Introduction to Kotlin Collections

Kotlin collections classes are the same as Java collections, but you can create them differently, and they also have an API that has been enhanced with Kotlin's extension functions.

Key: Mutable vs immutable collections

One important thing to know about Kotlin collections classes is that they're split into two categories:

- Mutable classes — elements can be added and removed
- Immutable classes — their elements can't be changed

In general I've found that my code is simpler if I prefer the immutable collections classes. That is, I always use an immutable class unless there's a compelling reason not to.

Array

Arrays are created with various `arrayOf` functions. Here are two ways to create arrays (implicit and explicit syntax):

```
val x = arrayOf(1,2,3)
val x: Array<Int> = arrayOf(1,2,3)

val y = arrayOf("a", "b", "c")
val y: Array<String> = arrayOf("a", "b", "c")
```

Array creation functions

Kotlin has these array-creation functions that are listed in *kotlin.Library.kt*:

```
fun <reified @PureReifiable T> arrayOfNulls(size: Int): Array<T?>
fun doubleArrayOf(vararg elements: Double): DoubleArray
fun floatArrayOf(vararg elements: Float): FloatArray
fun longArrayOf(vararg elements: Long): LongArray
fun intArrayOf(vararg elements: Int): IntArray
fun charArrayOf(vararg elements: Char): CharArray
fun shortArrayOf(vararg elements: Short): ShortArray
fun byteArrayOf(vararg elements: Byte): ByteArray
fun booleanArrayOf(vararg elements: Boolean): BooleanArray
```

Warning: Array is not the same as IntArray

Note: `arrayOf(1,2,3)` and `intArrayOf(1,2,3)` are not the same:

```
> val b: Array<Int> = arrayOf(1,2,3)
> b.javaClass
class [Ljava.lang.Integer;
> b.javaClass.name
[Ljava.lang.Integer;
> b.javaClass.kotlin
class kotlin.Array

> val c: IntArray = intArrayOf(1,2,3)
> c.javaClass
class [I
> c.javaClass.name
[I
> c.javaClass.kotlin
class kotlin.IntArray
```

TODO: Briefly discuss this.

The Kotlin List Class

This chapter introduces Kotlin lists. Some issues are:

- Mutable vs. immutable
- Handling null values
- Different types of lists

> Note: This chapter has a number of TODO items, but it currently shows how to create a variety of different list types.

Creation functions

Use these functions to create lists in Kotlin:

Function	Type
`arrayListOf`	`ArrayList<T>`
`emptyList`	`List<T>`
`listOf`	`List<T>`
`listOfNotNull`	`List<T>`
`mutableListOf`	`MutableList<T>`

Examples:

```
val list = listOf(1, 2, 3)
val list = arrayListOf(1, 2, 3)
val list = mutableListOf("a", "b", "c")

// empty lists
val list = listOf<Int>()
val list = arrayListOf<Double>()
val list = mutableListOf<String>()
val list: List<Int> = emptyList()

// nullability (implicit or explicit)
val list = listOf("a", null)                  // [a, null]
val list: List<String?> = listOf("a", null)   // [a, null]

val list = arrayListOf("a", null)             // [a, null]
val list = mutableListOf("a", null)           // [a, null]
val list = listOfNotNull<String>("a", null)   // [a]
val list = listOfNotNull("a", null)           // [a]

// comparison: listOf vs listOfNotNull
val list = listOf("a", null)                  // [a, null]
val list = listOfNotNull("a", null)           // [a]
```

Per the Kotlin docs, the difference between `listOf` and `listOfNotNull`:

- `listOf`: Returns an immutable list containing only the specified object element.
- `listOfNotNull`: Returns a new read-only list either of single given element, if it is not null, or empty list if the element is null.

Understanding casting

Some examples of casting and explicitly declaring list types:

```
val a: List<Int>                = listOf(1,2,3)
val b: Collection<Int>          = listOf(1,2,3)
val c: Iterable<Int>            = listOf(1,2,3)

val a: MutableList<Int>         = mutableListOf(1,2,3)
val b: List<Int>                = mutableListOf(1,2,3)
val c: MutableCollection<Int>   = mutableListOf(1,2,3)
val d: Collection<Int>          = mutableListOf(1,2,3)
val e: Iterable<Int>            = mutableListOf(1,2,3)
val f: MutableIterable<Int>     = mutableListOf(1,2,3)

val a: ArrayList<Int>           = arrayListOf(1,2,3)
val b: AbstractList<Int>        = arrayListOf(1,2,3)
val c: MutableList<Int>         = arrayListOf(1,2,3)
val d: AbstractCollection<Int>  = arrayListOf(1,2,3)
val e: MutableCollection<Int>   = arrayListOf(1,2,3)
val f: Collection<Int>          = arrayListOf(1,2,3)
val g: MutableIterable<Int>     = arrayListOf(1,2,3)
val h: Iterable<Int>            = arrayListOf(1,2,3)
val i: List<Int>                = arrayListOf(1,2,3)
```

The Kotlin Map Class

This chapter introduces maps in Kotlin.

Kotlin Map functions

Use these functions to create Maps in Kotlin:

Function	Type
`mapOf`	`Map<K,V>`
`hashMapOf`	`HashMap<K, V>`
`linkedMapOf`	`LinkedHashMap<K, V>`
`sortedMapOf`	`SortedMap<K, V>`
`mutableMapOf`	`MutableMap<K, V>`

Kotlin also has `linkedStringMapOf` and `stringMapOf` functions for JavaScript.

Examples:

```
val map = mapOf("b" to 2, "a" to 1)          // {b=2, a=1}
val map = hashMapOf("b" to 2, "a" to 1)      // {a=1, b=2}
val map = linkedMapOf("b" to 2, "a" to 1)    // {b=2, a=1}
val map = sortedMapOf("b" to 2, "a" to 1)    // {a=1, b=2}
val map = mutableMapOf("b" to 2, "a" to 1)   // {b=2, a=1}
```

Iterating over a map

Iterating over a map with `for`:

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)

for ((k,v) in map) {
     println("value of $k is $v")
}

// output
value of a is 1
value of b is 2
value of c is 3
```

Set

This chapter contains examples of how to create and use sets in Kotlin.

Creating sets

Use these functions to create sets:

Function	Type
`setOf`	`Set<T>`
`hashSetOf`	`HashSet<T>`
`linkedSetOf`	`LinkedHashSet<T>`
`sortedSetOf`	`TreeSet<T>`
`mutableSetOf`	`MutableSet<T>`

Kotlin also has `linkedStringSetOf` and `stringSetOf` functions for JavaScript.

Examples:

```
val set = setOf(3, 5, 1)          // [3, 5, 1]
val set = hashSetOf(3, 5, 1)      // [5, 1, 3]
val set = linkedSetOf(3, 5, 1)    // [3, 5, 1]
val set = sortedSetOf(3, 5, 1)    // [1, 3, 5]
val set = mutableSetOf(3, 5, 1)   // [3, 5, 1]
```

TODO

Need examples of:

- adding elements
- updating elements
- removing elements
- clearing the set
- iterating over (for-loop)

Collections Methods

Kotlin collections classes have many methods on them — *many*. To help make your life easier, this lesson shares examples for the most common methods that are available to sequential collections classes.

> Note: This lesson is very much a work in progress.

Filtering methods

binarySearch, distinct, distinctBy, drop, dropWhile, dropLast, dropLastWhile, elementAt, elementAtOrElse, elementAtOrNull, filter, filterIndexed, filterIsInstance, find, findLast, first, firstOrNull, get, getOrElse, indexOf, indexOfFirst, indexOfLast, intersect, last, lastIndexOf, lastOrNull, orEmpty, single, singleOrNull, take, takeWhile, takeLast, takeLastWhile, union

Transformers

associate, flatten, flatMap, intersect, map, mapNotNull, mapIndexed, mapIndexedNotNull, reversed, slice, sorted, sortedByDescending, sortedWith, union, unzip, zip

Aggregators

fold, foldIndexed, foldRight, foldRightIndexed, reduce, reduceIndexed, reduceRight, reduceRightIndexed

Grouping

groupBy, groupByTo, groupingBy, partition

Statistics

average, count, max, maxBy, maxWith, min, minBy, minWith, sum, sumBy

Information about the collection

all, any, contains, containsAll, none, forEach, forEachIndexed, isEmpty, isNotEmpty, onEach

Examples

The following examples use these lists:

```
val a = listOf(10, 20, 30, 40, 10)
val names = listOf("joel", "ed", "chris", "maurice")
```

Here are the examples:

```
a.any{it > 20}            //true
```

```
a.contains(10)             //true
a.count()                  //5
a.count{it > 10}           //3
a.distinct()                 //[10, 20, 30, 40]
a.distinctBy()
a.drop(1)                  //[20, 30, 40, 10]
a.drop(2)                  //[30, 40, 10]
a.dropLast(1)              //[10, 20, 30, 40]
a.dropLast(2)              //[10, 20, 30]
a.dropWhile{it < 30}       //[30, 40, 10]
a.dropLastWhile{it != 30}  //[10, 20, 30]
a.filter{it != 10}         //[20, 30, 40]
a.find{it != 10}           //20
a.first()                  //10
a.first{}
a.firstOrNull()            //TODO
a.fold(0){acc, x -> acc+x} //110 (sum function)
a.forEach{println(it)}     //prints out the list values
a.getOrElse(0){0}          //10
a.getOrElse(1){0}          //20
a.getOrElse(11){0}         //0
TODO: better groupBy
a.groupBy({it}, {it+1})    //{10=[11, 11], 20=[21], 30=[31], 40=[41]}
a.indexOf(10)              //0
a.indexOf(30)              //2
a.indexOfFirst()
a.indexOfLast()
a.intersect()
a.isEmpty()                //false
a.isNotEmpty()             //true
a.last()                   //10
a.last{}
a.lastIndexOf()
a.lastOrNull()
a.map{it + 1}              //[11, 21, 31, 41, 11]
a.map{it * 2}              //[20, 40, 60, 80, 20]
a.max()                    //40
a.maxBy{it + 3}            //40
maxWith(TODO)
a.min()                    //10
a.minBy{it + 3}            //10
minWith(TODO)
none
a.onEach{println(it)}      //prints each element and returns
                           //a copy of the list
orEmpty
a.partition{it >10}        //([20, 30, 40], [10, 10])
a.reduce{acc, x -> acc+x}  //110 (sum function)
a.slice(0..2)              //[10, 20, 30]
a.slice(1..2)              //[20, 30]
a.sorted()                 //[10, 10, 20, 30, 40]
a.sortedBy{it}             //[10, 10, 20, 30, 40]
names.sortedBy{it.length}  //[ed, joel, chris, maurice]
a.sortedWith()
a.sum()                    //110
a.sumBy{it + 1}            //115
a.take(1)                  //[10]
a.take(2)                  //[10, 20]
a.takeLast(1)              //[10]
a.takeLast(2)              //[40, 10]
a.takeLastWhile{it < 40}   //[10]
a.takeWhile{it < 40}       //[10, 20, 30]
a.union(names)             //[10, 20, 30, 40, joel, ed, chris, maurice]
a.zip(names)               //[(10, joel), (20, ed), (30, chris), (40, maurice)]
names.zip(a)               //[(joel, 10), (ed, 20), (chris, 30), (maurice, 40)]
```

Convert a list to a String

Convert a list/array to a `String`:

```
val nums = listOf(1,2,3,4,5)

> nums.joinToString()
1, 2, 3, 4, 5

> nums.joinToString(
    separator = ", ",
    prefix = "[",
    postfix = "]",
    limit = 3,
    truncated = "there’s more ..."
)
[1, 2, 3, there’s more ...]
```

Common Map Methods

This lesson is TODO.

TODO

This chapter is TODO.

Kotlin and Swing

Kotlin works with Java Swing classes like `JFrame`, `JTextArea`, etc., very easily. Here's an example of a Kotlin application that opens a `JFrame`, adds a few components to it, and then displays it:

```
import java.awt.BorderLayout
import java.awt.Dimension
import javax.swing.JFrame
import javax.swing.JScrollPane
import javax.swing.JTextArea

fun main(args: Array<String>) {

    val textArea = JTextArea()
    textArea.setText("Hello, Kotlin/Swing world")
    val scrollPane = JScrollPane(textArea)

    val frame = JFrame("Hello, Kotlin/Swing")
    frame.getContentPane().add(scrollPane, BorderLayout.CENTER)
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
    frame.setSize(Dimension(600, 400))
    frame.setLocationRelativeTo(null)
    frame.setVisible(true)

}
```

To see how that code works, save it to a file named *KotlinSwing.kt*, then compile it:

```
$ kotlinc KotlinSwing.kt -include-runtime -d KotlinSwing.jar
```

then execute the jar file with this `java` command:

```
$ java -jar KotlinSwing.jar
```

Kotlin Build Tools

- TODO: Ant
- TODO: Gradle
- TODO: Maven
- TODO: other

Kotlin Idioms (TODO)

- Don’t use null
- Don’t throw exceptions
- Use nullability
- Pure functions and EOP

A Kotlin OOP Example

This lesson is TODO.

Kotlin at the Command Line

This lesson demonstrates examples of how to use Kotlin commands at the command line, including `kotlinc`.

Compiling a Kotlin file

```
$ kotlinc Hello.kt
```

That command creates a file named *HelloKt.class*.

Compiling to an executable Jar file

Compiling to an executable Jar file:

```
$ kotlinc Hello.kt -include-runtime -d Hello.jar
```

Then execute the jar file with this `java` command:

```
$ java -jar Hello.jar
Hello, world
```

Android

This section of the book will contain Kotlin syntax examples that can help in building Android applications.

FloatingActionButton lambda

This code:

```
fab.setOnClickListener(object : View.OnClickListener {
    override fun onClick(view: View) {
        save("Note1.txt")
    }
})
```

can be written like this:

```
fab.setOnClickListener { save("Note1.txt") }
```

Contributors

The initial version of this book is created by Alvin Alexander, who has previously written these books:

- Scala Cookbook
- Functional Programming, Simplified
- Hello, Scala
- A Survival Guide for New Consultants
- How I Sold My Business: A Personal Diary

License

Everything in this book — **with the exception of the book cover** — is released under the Attribution-ShareAlike 4.0 International license. The license is described in full detail below and at that URL.

For those viewing a print version of this book, here are those URLs:

- Alvin Alexander's website: https://alvinalexander.com
- The *Kotlin Quick Reference* website: http://kotlin-quick-reference.com
- The Attribution-ShareAlike 4.0 International license: https://creativecommons.org/licenses/by-sa/4.0/)

The cover image

If you want to fork the project that's fine, but it doesn't make sense to have the same cover image on different books, so that's one reason why the cover image is not open source. That, and I paid a designer to create it and a stock image company for part of the artwork. So, if you fork the project, create your own cover.

> I wouldn't include the image in this repository if I didn't have to, but Gitbook requires it.

If you're interested in a good designer, Alex Blokowsky on 99designs.com created this cover image for me.

The license

The text of the Attribution-ShareAlike 4.0 International license follows.

```
Attribution-ShareAlike 4.0 International
=======================================

Creative Commons Corporation ("Creative Commons") is not a law firm and
does not provide legal services or legal advice. Distribution of
Creative Commons public licenses does not create a lawyer-client or
other relationship. Creative Commons makes its licenses and related
information available on an "as-is" basis. Creative Commons gives no
warranties regarding its licenses, any material licensed under their
terms and conditions, or any related information. Creative Commons
disclaims all liability for damages resulting from their use to the
fullest extent possible.

Using Creative Commons Public Licenses

Creative Commons public licenses provide a standard set of terms and
conditions that creators and other rights holders may use to share
original works of authorship and other material subject to copyright
and certain other rights specified in the public license below. The
following considerations are for informational purposes only, are not
exhaustive, and do not form part of our licenses.

     Considerations for licensors: Our public licenses are
     intended for use by those authorized to give the public
     permission to use material in ways otherwise restricted by
     copyright and certain other rights. Our licenses are
     irrevocable. Licensors should read and understand the terms
     and conditions of the license they choose before applying it.
     Licensors should also secure all rights necessary before
     applying our licenses so that the public can reuse the
     material as expected. Licensors should clearly mark any
     material not subject to the license. This includes other CC-
     licensed material, or material used under an exception or
```

```
     limitation to copyright. More considerations for licensors:
    wiki.creativecommons.org/Considerations_for_licensors

     Considerations for the public: By using one of our public
     licenses, a licensor grants the public permission to use the
     licensed material under specified terms and conditions. If
     the licensor's permission is not necessary for any reason--for
     example, because of any applicable exception or limitation to
     copyright--then that use is not regulated by the license. Our
     licenses grant only permissions under copyright and certain
     other rights that a licensor has authority to grant. Use of
     the licensed material may still be restricted for other
     reasons, including because others have copyright or other
     rights in the material. A licensor may make special requests,
     such as asking that all changes be marked or described.
     Although not required by our licenses, you are encouraged to
     respect those requests where reasonable. More_considerations
     for the public:
    wiki.creativecommons.org/Considerations_for_licensees

=======================================================================

Creative Commons Attribution-ShareAlike 4.0 International Public
License

By exercising the Licensed Rights (defined below), You accept and agree
to be bound by the terms and conditions of this Creative Commons
Attribution-ShareAlike 4.0 International Public License ("Public
License"). To the extent this Public License may be interpreted as a
contract, You are granted the Licensed Rights in consideration of Your
acceptance of these terms and conditions, and the Licensor grants You
such rights in consideration of benefits the Licensor receives from
making the Licensed Material available under these terms and
conditions.


Section 1 -- Definitions.

  a. Adapted Material means material subject to Copyright and Similar
     Rights that is derived from or based upon the Licensed Material
     and in which the Licensed Material is translated, altered,
     arranged, transformed, or otherwise modified in a manner requiring
     permission under the Copyright and Similar Rights held by the
     Licensor. For purposes of this Public License, where the Licensed
     Material is a musical work, performance, or sound recording,
     Adapted Material is always produced where the Licensed Material is
     synched in timed relation with a moving image.

  b. Adapter's License means the license You apply to Your Copyright
     and Similar Rights in Your contributions to Adapted Material in
     accordance with the terms and conditions of this Public License.

  c. BY-SA Compatible License means a license listed at
     creativecommons.org/compatiblelicenses, approved by Creative
     Commons as essentially the equivalent of this Public License.

  d. Copyright and Similar Rights means copyright and/or similar rights
     closely related to copyright including, without limitation,
     performance, broadcast, sound recording, and Sui Generis Database
     Rights, without regard to how the rights are labeled or
     categorized. For purposes of this Public License, the rights
     specified in Section 2(b)(1)-(2) are not Copyright and Similar
     Rights.

  e. Effective Technological Measures means those measures that, in the
     absence of proper authority, may not be circumvented under laws
     fulfilling obligations under Article 11 of the WIPO Copyright
     Treaty adopted on December 20, 1996, and/or similar international
     agreements.
```

```
  f. Exceptions and Limitations means fair use, fair dealing, and/or
     any other exception or limitation to Copyright and Similar Rights
     that applies to Your use of the Licensed Material.

  g. License Elements means the license attributes listed in the name
     of a Creative Commons Public License. The License Elements of this
     Public License are Attribution and ShareAlike.

  h. Licensed Material means the artistic or literary work, database,
     or other material to which the Licensor applied this Public
     License.

  i. Licensed Rights means the rights granted to You subject to the
     terms and conditions of this Public License, which are limited to
     all Copyright and Similar Rights that apply to Your use of the
     Licensed Material and that the Licensor has authority to license.

  j. Licensor means the individual(s) or entity(ies) granting rights
     under this Public License.

  k. Share means to provide material to the public by any means or
     process that requires permission under the Licensed Rights, such
     as reproduction, public display, public performance, distribution,
     dissemination, communication, or importation, and to make material
     available to the public including in ways that members of the
     public may access the material from a place and at a time
     individually chosen by them.

  l. Sui Generis Database Rights means rights other than copyright
     resulting from Directive 96/9/EC of the European Parliament and of
     the Council of 11 March 1996 on the legal protection of databases,
     as amended and/or succeeded, as well as other essentially
     equivalent rights anywhere in the world.

  m. You means the individual or entity exercising the Licensed Rights
     under this Public License. Your has a corresponding meaning.


Section 2 -- Scope.

  a. License grant.

       1. Subject to the terms and conditions of this Public License,
          the Licensor hereby grants You a worldwide, royalty-free,
          non-sublicensable, non-exclusive, irrevocable license to
          exercise the Licensed Rights in the Licensed Material to:

            a. reproduce and Share the Licensed Material, in whole or
               in part; and

            b. produce, reproduce, and Share Adapted Material.

       2. Exceptions and Limitations. For the avoidance of doubt, where
          Exceptions and Limitations apply to Your use, this Public
          License does not apply, and You do not need to comply with
          its terms and conditions.

       3. Term. The term of this Public License is specified in Section
          6(a).

       4. Media and formats; technical modifications allowed. The
          Licensor authorizes You to exercise the Licensed Rights in
          all media and formats whether now known or hereafter created,
          and to make technical modifications necessary to do so. The
          Licensor waives and/or agrees not to assert any right or
          authority to forbid You from making technical modifications
          necessary to exercise the Licensed Rights, including
          technical modifications necessary to circumvent Effective
```

```
           Technological Measures. For purposes of this Public License,
           simply making modifications authorized by this Section 2(a)
           (4) never produces Adapted Material.

       5. Downstream recipients.

            a. Offer from the Licensor -- Licensed Material. Every
               recipient of the Licensed Material automatically
               receives an offer from the Licensor to exercise the
               Licensed Rights under the terms and conditions of this
               Public License.

            b. Additional offer from the Licensor -- Adapted Material.
               Every recipient of Adapted Material from You
               automatically receives an offer from the Licensor to
               exercise the Licensed Rights in the Adapted Material
               under the conditions of the Adapter's License You apply.

            c. No downstream restrictions. You may not offer or impose
               any additional or different terms or conditions on, or
               apply any Effective Technological Measures to, the
               Licensed Material if doing so restricts exercise of the
               Licensed Rights by any recipient of the Licensed
               Material.

       6. No endorsement. Nothing in this Public License constitutes or
          may be construed as permission to assert or imply that You
          are, or that Your use of the Licensed Material is, connected
          with, or sponsored, endorsed, or granted official status by,
          the Licensor or others designated to receive attribution as
          provided in Section 3(a)(1)(A)(i).

  b. Other rights.

       1. Moral rights, such as the right of integrity, are not
          licensed under this Public License, nor are publicity,
          privacy, and/or other similar personality rights; however, to
          the extent possible, the Licensor waives and/or agrees not to
          assert any such rights held by the Licensor to the limited
          extent necessary to allow You to exercise the Licensed
          Rights, but not otherwise.

       2. Patent and trademark rights are not licensed under this
          Public License.

       3. To the extent possible, the Licensor waives any right to
          collect royalties from You for the exercise of the Licensed
          Rights, whether directly or through a collecting society
          under any voluntary or waivable statutory or compulsory
          licensing scheme. In all other cases the Licensor expressly
          reserves any right to collect such royalties.


Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the
following conditions.

  a. Attribution.

       1. If You Share the Licensed Material (including in modified
          form), You must:

            a. retain the following if it is supplied by the Licensor
               with the Licensed Material:

                 i. identification of the creator(s) of the Licensed
                    Material and any others designated to receive
                    attribution, in any reasonable manner requested by
```

```
                    the Licensor (including by pseudonym if
                    designated);

                ii. a copyright notice;

               iii. a notice that refers to this Public License;

                iv. a notice that refers to the disclaimer of
                    warranties;

                 v. a URI or hyperlink to the Licensed Material to the
                    extent reasonably practicable;

            b. indicate if You modified the Licensed Material and
               retain an indication of any previous modifications; and

            c. indicate the Licensed Material is licensed under this
               Public License, and include the text of, or the URI or
               hyperlink to, this Public License.

       2. You may satisfy the conditions in Section 3(a)(1) in any
          reasonable manner based on the medium, means, and context in
          which You Share the Licensed Material. For example, it may be
          reasonable to satisfy the conditions by providing a URI or
          hyperlink to a resource that includes the required
          information.

       3. If requested by the Licensor, You must remove any of the
          information required by Section 3(a)(1)(A) to the extent
          reasonably practicable.

  b. ShareAlike.

     In addition to the conditions in Section 3(a), if You Share
     Adapted Material You produce, the following conditions also apply.

       1. The Adapter's License You apply must be a Creative Commons
          license with the same License Elements, this version or
          later, or a BY-SA Compatible License.

       2. You must include the text of, or the URI or hyperlink to, the
          Adapter's License You apply. You may satisfy this condition
          in any reasonable manner based on the medium, means, and
          context in which You Share Adapted Material.

       3. You may not offer or impose any additional or different terms
          or conditions on, or apply any Effective Technological
          Measures to, Adapted Material that restrict exercise of the
          rights granted under the Adapter's License You apply.


Section 4 -- Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that
apply to Your use of the Licensed Material:

  a. for the avoidance of doubt, Section 2(a)(1) grants You the right
     to extract, reuse, reproduce, and Share all or a substantial
     portion of the contents of the database;

  b. if You include all or a substantial portion of the database
     contents in a database in which You have Sui Generis Database
     Rights, then the database in which You have Sui Generis Database
     Rights (but not its individual contents) is Adapted Material,

     including for purposes of Section 3(b); and
  c. You must comply with the conditions in Section 3(a) if You Share
     all or a substantial portion of the contents of the database.
```

```
For the avoidance of doubt, this Section 4 supplements and does not
replace Your obligations under this Public License where the Licensed
Rights include other Copyright and Similar Rights.


Section 5 -- Disclaimer of Warranties and Limitation of Liability.

  a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE
     EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS
     AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF
     ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS,
     IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION,
     WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR
     PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS,
     ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT
     KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT
     ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

  b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE
     TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION,
     NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT,
     INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES,
     COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR
     USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN
     ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR
     DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR
     IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.

  c. The disclaimer of warranties and limitation of liability provided
     above shall be interpreted in a manner that, to the extent
     possible, most closely approximates an absolute disclaimer and
     waiver of all liability.


Section 6 -- Term and Termination.

  a. This Public License applies for the term of the Copyright and
     Similar Rights licensed here. However, if You fail to comply with
     this Public License, then Your rights under this Public License
     terminate automatically.

  b. Where Your right to use the Licensed Material has terminated under
     Section 6(a), it reinstates:

       1. automatically as of the date the violation is cured, provided
          it is cured within 30 days of Your discovery of the
          violation; or

       2. upon express reinstatement by the Licensor.

     For the avoidance of doubt, this Section 6(b) does not affect any
     right the Licensor may have to seek remedies for Your violations
     of this Public License.

  c. For the avoidance of doubt, the Licensor may also offer the
     Licensed Material under separate terms or conditions or stop
     distributing the Licensed Material at any time; however, doing so
     will not terminate this Public License.

  d. Sections 1, 5, 6, 7, and 8 survive termination of this Public
     License.


Section 7 -- Other Terms and Conditions.

  a. The Licensor shall not be bound by any additional or different
     terms or conditions communicated by You unless expressly agreed.

  b. Any arrangements, understandings, or agreements regarding the
```

```
      Licensed Material not stated herein are separate from and
      independent of the terms and conditions of this Public License.


Section 8 -- Interpretation.

  a. For the avoidance of doubt, this Public License does not, and
     shall not be interpreted to, reduce, limit, restrict, or impose
     conditions on any use of the Licensed Material that could lawfully
     be made without permission under this Public License.

  b. To the extent possible, if any provision of this Public License is
     deemed unenforceable, it shall be automatically reformed to the
     minimum extent necessary to make it enforceable. If the provision
     cannot be reformed, it shall be severed from this Public License
     without affecting the enforceability of the remaining terms and
     conditions.

  c. No term or condition of this Public License will be waived and no
     failure to comply consented to unless expressly agreed to by the
     Licensor.

  d. Nothing in this Public License constitutes or may be interpreted
     as a limitation upon, or waiver of, any privileges and immunities
     that apply to the Licensor or You, including from the legal
     processes of any jurisdiction or authority.


=======================================================================

Creative Commons is not a party to its public
licenses. Notwithstanding, Creative Commons may elect to apply one of
its public licenses to material it publishes and in those instances
will be considered the “Licensor.” The text of the Creative Commons
public licenses is dedicated to the public domain under the CC0 Public
Domain Dedication. Except for the limited purpose of indicating that
material is shared under a Creative Commons public license or as
otherwise permitted by the Creative Commons policies published at
creativecommons.org/policies, Creative Commons does not authorize the
use of the trademark "Creative Commons" or any other trademark or logo
of Creative Commons without its prior written consent including,
without limitation, in connection with any unauthorized modifications
to any of its public licenses or any other arrangements,
understandings, or agreements concerning use of licensed material. For
the avoidance of doubt, this paragraph does not form part of the
public licenses.

Creative Commons may be contacted at creativecommons.org.
```

About This Book

Kotlin Quick Reference is created by Alvin Alexander, author of:

- Scala Cookbook
- Functional Programming, Simplified
- Hello, Scala

Gratitude: Many thanks to the creators of the Kotlin programming language for creating such a wonderful programming language. Additional thanks to the creators of GitBook, who made it easy to create this book online and in multiple other electronic formats.