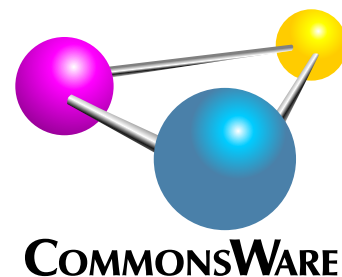


FINAL Version

Elements of Kotlin

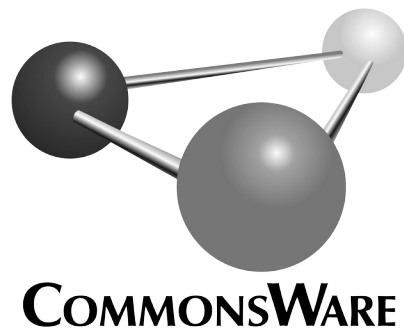


Mark L. Murphy



Elements of Kotlin

by Mark L. Murphy



Elements of Kotlin
by Mark L. Murphy

Copyright © 2018-2021 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

Printing History:
December 2021: FINAL Version

The CommonsWare name and logo, “Busy Coder's Guide”, and related trade dress are trademarks of CommonsWare, LLC.

All other trademarks referenced in this book are trademarks of their respective firms.

The publisher and author(s) assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

- [Preface](#)
 - Prerequisites vii
 - Source Code and Its License viii
 - Acknowledgments viii
- [Introducing Kotlin](#)
 - Why? 1
 - ...And Why Not? 1
 - What You Need to Know 2
 - Look At All the Kotlin! 4
 - Getting Kotlin, Normally 5
 - Introducing the Klassbook 6
 - How This Book is Structured 6
- [A Few “Hello, World!” Examples](#)
 - Just the Statement 9
 - Wrapped in a Function 10
 - Wrapped in a Class 11
 - So, What Does This Do? 11
 - Using the Klassbook 13
 - Running These Snippets in an IDE 14
- [Basic Types and Expressions](#)
 - Basic Types and “Objectness” 17
 - Numbers 17
 - Boolean 20
 - Strings 20
 - Characters 22
 - A Quick Note About Equality 23
- [Variables... Whether They Vary or Not](#)
 - Declaring Variables 25
 - Declaring Read-Only Variables 26
 - Prefer val Over var 28
 - String Interpolation 28
 - More Operators 29
 - No Automatic Number Conversions 31
 - Hey, What About ? 32
- [Functions](#)

◦ Functions with Parameters	35
◦ Functions with Return Types	36
◦ Local Variables	37
◦ Fancier Functions	37
• Collections and Lambdas	
◦ Major Collection Types and Creation Functions	43
◦ Basic Usage	47
◦ Immutability and Collections	49
◦ Introducing Lambda Expressions	50
◦ Common Collection Operations	52
◦ Varargs	56
◦ Other JVM Collections	58
◦ get() and [] Syntax	59
• If, When, and While	
◦ If	61
◦ When	63
◦ While	67
◦ If/When As Expressions	68
◦ Bustin' Out	71
• Basic Classes	
◦ Basic Classes	77
◦ Creating Instances of Classes	77
◦ Packages	78
◦ Common Contents	79
◦ this	81
◦ Constructors	82
◦ Inheritance	88
• Comments and Documentation	
◦ Basic Comment Syntax	105
◦ Introducing KDoc	106
• Properties	
◦ Initialization	112
◦ Constants	117
◦ Getting Down in the Weeds	117
• Visibility and Scope	
◦ Visibility	119
◦ Scope	124
• Abstract Classes and Interfaces	
◦ The Objective: Contracts	127
◦ Abstract Classes	128
◦ Interfaces	130

- Which Do You Use? 136
- [Data Class](#)
 - How You Declare It 139
 - What You Gain 139
 - What You Lose 142
 - Data Classes with Other Properties 142
- [The object Keyword](#)
 - Singletons 145
 - Companion Objects 147
 - Nested Objects 149
 - Object Expressions 149
- [Nested Objects and Classes](#)
 - Nested Objects 153
 - Nested Classes 155
 - Nested Interfaces and Abstract Classes 160
- [Enums and Sealed Classes](#)
 - Enums 163
 - Sealed Classes 170
- [Generics](#)
 - OK, What Are These For Again? 179
 - Instantiation with Generics 180
 - Applying Generics to Classes and Interfaces 185
 - Upper Bounds 187
 - Generics, WTF? 188
- [Exceptions](#)
 - Catching Exceptions 189
 - Raising Exceptions 194
 - Defining Exceptions 195
 - Checked vs. Unchecked Exceptions 196
- [Annotations](#)
 - Where Annotations Come From 199
 - Applying Annotations 200
 - Defining Annotations 205
- [Nullability](#)
 - Introducing Nullable Types 209
 - Expressions with Nullable Types 210
 - Nullable Types and Generics 218
 - Nullable Types and Casts 219
 - Objective: Minimize Nulls 220
- [Scope Functions](#)
 - let() 223

◦ apply()	225
◦ run()	228
◦ with()	229
◦ also()	230
◦ use()	231
◦ Summary	231
• Functional Programming	
◦ Your App Might Not Be Functional	233
◦ Where Immutability Comes Into Play	234
◦ Examples of Functional Kotlin	235
◦ Function Types	241
◦ Arrays, Collections,... And Sequences	250
• Extension Functions	
◦ The Case of the Utility Function	256
◦ Monkeying Around	258
◦ Declaring Extension Functions	259
◦ Calling Extension Functions	261
◦ The Limitations	261
• Java Interoperability	
◦ Recap of Interoperability	263
◦ Kotlin Calling Java	264
◦ Java Calling Kotlin	274
◦ Decompiling Kotlin to Java	287
• Changes in Newer Kotlin Versions	
◦ Version 1.3	289
◦ Version 1.4	292
• Custom Accessors	
◦ Defining Custom Accessors	299
◦ Anti-Patterns	301
◦ Alternative: Delegates	302
• Extension Properties	
◦ Recapping Extension Functions	303
◦ Recapping Property Custom Accessors	303
◦ Extension Properties = A Mashup	304
• Escaping Keywords	
◦ The Scenario: Mockito	307
◦ The Problem: Keywords	307
◦ The Solution: Backticks	308
• Escaped Method Names	
◦ The Scenario: JUnit Tests	309
◦ The Problem: You Are Tired of CamelCase	309

- The Solution: Backticks 310
- [Property Delegates](#)
 - A Refresher on lazy 311
 - What’s Really Going On 312
 - Other Stock Property Delegates 313
- [Class Delegation](#)
 - Playing Favorites 317
 - Manual Delegation 319
 - The Class Delegate Alternative 320
- [Constants](#)
 - Declaring a Constant 321
 - Why Bother, When We Have val? 321
 - Constant Type Limitations 322
- [Abstract Properties](#)
 - But, But, But... Why? 323
 - Abstract val 323
 - Abstract var 325
- [Covariance in Generics](#)
 - You Can’t Put That in That! 327
 - out Is a Direction 328
 - Declaration-Site and Use-Site Variance 329
- [Contravariance in Generics](#)
 - Declaration-Site and Use-Site Variance 335
- [Anonymous Functions](#)
 - The Problem: Return Types 339
 - The Solution: Anonymous Function Syntax 340
 - Constraints and Effects 340
- [Local Functions](#)
 - I Heard You Like Functions... 343
 - ...So I Put a Function in Your Function 344
 - Funception 345
- [Local Types](#)
 - Scenario: Perils of Pair Programming 347
- [Inline Functions](#)
 - Macro History 351
 - Inline Functions: Like Macros 352
 - Inline Properties 353
- [Inline Classes](#)
 - What? 355
 - Why? 355
 - Alpha! 357

- [Reified Type Parameters](#)
 - Type Erasure (Other Than Via the Backspace Key) 359
 - Reified = Retained Type for Inline Functions 360
- [noinline and crossinline](#)
 - noinline: Keep the Lambda as an Object 363
 - crossinline: Allowing return 365
- [Receivers in Function Types](#)
 - What with() Looks Like 368
 - What apply() Looks Like 369
 - Use Case: DSL 369
- [Renamed Imports](#)
 - The Scenario: Observable 371
 - The Problem: Import Name Collisions 372
 - The Solution: import ... as 372
- [Operator Overloading](#)
 - What the #\$_&%!? 375
 - The Concept of Operator Overloading 375
 - Example: Dividing a String 376
 - So. Many. Operators. 376
- [Infix Functions](#)
 - Postfix and Infix 379
 - The infix Keyword 379
 - Limitations 380
- [Destructuring Declarations](#)
 - The Components of a Class 381
 - OK, Why Would We Use This? 382
 - So, Why Does This Exist? 383
- [Labeled Returns](#)
 - Where return Goes By Default 385
 - Returning Just from a Lambda 387
 - Applying a Label 387
 - Is Any of This a Good Idea? 388
- [Nothing](#)
 - Nothing: It's On the Bottom 390
 - Uses of Nothing 390
- [Types of Keywords](#)
 - Hard Keywords 395
 - Soft Keywords 396
 - Modifier Keywords 397
 - In General, Avoid Keywords 398

Preface

Thanks!

First, thanks for your interest in Kotlin! Kotlin is one of the fastest-growing programming languages in the world. In no small part that is due to Google's endorsement of Kotlin for Android app development, as Android is the world's most widely-used operating system. Perhaps you are interested in using Kotlin for writing Android apps. Perhaps you are interested in Kotlin in other places, such as for Web apps. Kotlin itself is the same no matter where you apply it, so this book can help you get up to speed with the syntax and patterns that you will see as you read other developers' Kotlin code and start writing it yourself.

Also, thanks for reading this book! This is one of a series of books published by [CommonsWare](https://commonsware.com/), each designed to help a developer become proficient in some piece of technology.

Prerequisites

This book is written for newcomers in Kotlin. If you have already experimented some with Kotlin, you might elect to skim the first few chapters, until you start encountering topics that are new to you.

This book does expect you to have prior programming experience. A lot of comparisons will be made to Java, as Java is a very popular programming language and the one that was dominant for the first decade of Android app development. You do not need to be a Java programmer to use this book, though.

Source Code and Its License

The source code in this book is licensed under the [Apache 2.0 License](#), in case you have the desire to reuse any of it.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Acknowledgments

The author would like to thank JetBrains, the firm that created Kotlin. Without them, this book would not be possible.

Of course, without them, this book would be really strange, as Kotlin would not exist, so this book would be trying to explain how to use a non-existent programming language.

Kotlin Basics

Introducing Kotlin

Kotlin's popularity exploded in 2017, when Google announced official support for Kotlin development for Android apps. Whether you are looking to use Kotlin for Android app development or for other sorts of projects, you have chosen one of the fastest-growing languages and one with a tremendous amount of “buzz”.

This book will help you get up to speed on the core aspects of Kotlin, plus help you understand unusual Kotlin syntax that you will encounter from time to time.

Why?

Kotlin is most often compared to Java. Kotlin was originally created to serve as a Java replacement, though [that story has gotten somewhat more complicated](#). Also, Java was the preeminent language for Android app development, and so many developers are coming to Kotlin having had some amount of Java experience.

Compared to Java, Kotlin has less “ceremony”. It is designed to allow code to be *concise*, with fewer characters used to express the same programming algorithms. For example, Java and many other programming languages (e.g., C, C++) use a semicolon (;) to designate the end of a statement. In Kotlin, the semicolon usually is optional. That is a small change in syntax, but Kotlin is full of these sorts of smaller changes, each providing you with more results with less actual code.

...And Why Not?

On the other hand, Kotlin is one of the most complex programming languages ever created. For most code, Kotlin can be concise. However, in many cases, that is because the “real code” is hidden, supplied by standard libraries. The programming

techniques that allow those libraries to let you write concise Kotlin code are also available to you, and that is where the complexity lies.

Fortunately, most of that complexity can be ignored at the outset. That complexity exists for “power users” of Kotlin, such as people developing other Kotlin libraries. They too can use those techniques to make their libraries easy to consume. However, many developers will never examine the code of such a library, let alone create one, and so for many developers, this complexity is something that will be encountered only rarely.

Kotlin also lends itself towards *terse* programming. “Concise” is “short but understandable”, while “terse” is “short, but perhaps not understandable”. Keeping Kotlin code short but still readable, particularly by newcomers, is the sort of thing that some Kotlin developers ignore in a drive to minimize the number of characters in their Kotlin source code.

What You Need to Know

This book assumes that you have some amount of existing programming experience in an object-oriented programming language. To help explain certain Kotlin syntax, the book will draw parallels to Java, JavaScript, and Ruby, three popular programming languages. While you do not need experience in any of those, you will need to know the basics of object-oriented programming.

The following sections outline the sorts of prior experience that you need to have in order to make the best use of this book.

Data Types and Expressions

Nearly every programming language has the concept of data types: strings, numbers, booleans, and so on. The exact roster of data types varies by language, as do some of the details of how those data types are implemented. But, you should be comfortable in thinking about how strings, numbers, and similar types of data flow through our programming.

Often, these pieces of data are used in calculations, forming parts of expressions to derive new data. That could be mathematical expressions, such as $2 + 2$. That could be expressions tied to other data types, such as string concatenation (e.g., “foo” + “bar”). The details of what expressions are possible and their syntax varies a bit by programming language, but all languages can do this sort of thing.

Objects and Classes

Kotlin is a class-based object-oriented language. Quoting [Wikipedia](#):

The most popular and developed model of OOP is a class-based model, as opposed to an object-based model. In this model, objects are entities that combine state (i.e. data), behavior (i.e. procedures, or methods) and identity (unique existence among all other objects). The structure and behavior of an object are defined by a class, which is a definition, or blueprint, of all objects of a specific type. An object must be explicitly created based on a class and an object thus created is considered to be an instance of that class.

Java and Ruby are class-based object-oriented languages, where we use the `class` keyword to begin the description of a class. JavaScript originally was not a class-based language, though recent updates have pulled it much closer to its class-based counterparts.

Methods or Functions

In a class-based object-oriented language, we need to tell the class what its behaviors are. A `Box` class might have `open()` and `close()` behaviors, for example.

In Java and Ruby, those behaviors would be implemented in the form of “methods”. In Kotlin and JavaScript, those behaviors are called “functions”.

Fields, Properties, and Variables

Similarly, in a class-based object-oriented language, we need to tell the class what data it has to work on. A `Box` class might have:

- Its contents (representing what the box holds)
- Its material (representing what the box itself is made of: cardboard, wood, steel, adamantium, etc.)
- Its `isClosed` state
- Its length, width, and height
- And so on

The term for this varies widely:

- Java refers to them as “fields”

- Ruby refers to them as “instance variables”
- JavaScript and Kotlin refer to them as “properties”

Also, our methods or functions will have temporary holding spots for bits of data. The input is usually referred to as “parameters” or “arguments”, so a `seal()` function might accept a parameter indicating how the box should be sealed (with tape, with staples, with welds, etc.). The working data inside the function is usually called “variables”, so a `seal()` function might use the `length`, `width`, and `height` properties to calculate how much tape is needed to seal the box, holding that calculated value in a variable.

Look At All the Kotlin!

This book is focused on Kotlin syntax, so you understand how to read and write Kotlin code. However, usually, if you are writing Kotlin code, the point behind the code is not for it to just sit there. Most likely, you plan to run that code.

Programming languages sometimes have a variety of ways that their code can be run. For example, Ruby code can be run on a few Ruby interpreters, or on the Java Virtual Machine (JVM) via JRuby, or possibly compiled into native code. Of those options, the standard Ruby interpreter is the most common, and is the one that people tend to think of. Similarly, despite NodeJS’s popularity, when a lot of people think about running JavaScript code, they will think of doing so inside of a Web browser.

Similarly, Kotlin has a few “runtime environments” to consider.

Kotlin/JVM

If you just hear the name “Kotlin”, without any qualifiers, it is likely that the person in question is referring to the original form of Kotlin, which ran on the JVM. Kotlin’s compiler would create Java bytecode from the Kotlin source code, just as Java’s compiler creates Java bytecode from Java source code. The resulting bytecode can be used by `java` and similar tools, regardless of whether it was written in Java, Kotlin, or something else.

In particular, if you are interested in Android app development, when you think of Kotlin, most likely you are thinking of Kotlin being compiled in an Android project. In that case, while Android does not use the JVM, the same basic mechanism is applied: Kotlin’s compiler generates Java bytecode, which the Android build process

then converts into Dalvik bytecode for use in Android apps.

Kotlin/JS

For the first few years of Kotlin's development, Kotlin simply was Kotlin/JVM.

In 2014, Kotlin/JS was introduced. This allows Kotlin's compiler to "compile" Kotlin source code into JavaScript source code, with an eye towards that code being used in an environment like NodeJS. The JavaScript that you get from the compilation process is not necessarily going to be easily readable by ordinary humans, but that's not the goal. The goal is to be able to write in Kotlin and run it in a traditional JavaScript runtime environment.

Kotlin/Native

More recently, the Kotlin team has added Kotlin/Native. This compiles Kotlin code to native opcodes, the same as you might find with a C/C++ compiler. In particular, Kotlin/Native is designed to generate code that can run on iOS and macOS, interoperating with Objective-C.

Kotlin/Common

Kotlin/Common does not represent a new runtime environment for Kotlin, at least in the strict sense. Kotlin/Common is the term given to a type of library that limits itself to using Kotlin classes and functions that exist for all Kotlin runtime environments. Such a library can be used by a Kotlin/JVM project as easily as by a Kotlin/JS or Kotlin/Native project, for example.

Kotlin/Multiplatform

Kotlin/Multiplatform also is not a new runtime environment for Kotlin. Rather, it is a way of setting up a project such that some core app logic is written in a Kotlin/Common library, while other "modules" are set up for specific runtime environments. Overall, the entire project can target 2+ runtime environments, such as using Kotlin as the foundation for both an Android (Kotlin/JVM) and iOS (Kotlin/Native) app.

Getting Kotlin, Normally

For many developers, you will not need to download and install a Kotlin

environment, as that will be handled by your IDE or other build tools.

For example, Android Studio users get Kotlin “for free” as part of building Android Studio-based projects. Kotlin’s toolchain — compilers, etc. — are obtained in the form of Gradle plugins, coupled with some Kotlin support “baked into” Android Studio itself. Similarly, the Kotlin runtime bits are added via Gradle to an Android project. Plugins are available for IntelliJ IDEA, Eclipse, and other IDEs.

However, you can also [download a standalone Kotlin compiler](#) if desired.

Introducing the Klassbook

An easy way to experiment with a programming language is to use a REPL.

REPL is an acronym, describing a tool that will **R**ead typed-in source code, **E**valuate it, **P**rint the results of that evaluation in the tool, and **L**oop back to process more input. Frequently, a REPL resembles an operating system shell or terminal, such as Windows’ Command Prompt or a bash shell in macOS or Linux.

Using a REPL gives you rapid feedback in a low-effort environment. You do not need a full IDE or a full application project to experiment with bits of language syntax and see how they work.

[Klassbook](#) is a REPL set up for use with this book. Most of the code examples shown in this book have corresponding Klassbook pages, where you can see the code, run the code to see the results, and even edit the code to try your own experiments. We will see how to use the Klassbook more in [the next chapter](#).

How This Book is Structured

This book has two objectives:

- Help you start writing basic Kotlin code
- Help you start reading advanced Kotlin code

Many developers do not need to be writing advanced Kotlin code, but developers will see fancy Kotlin syntax from time to time (conference presentations, blog posts, etc.). Even those who will be writing fairly basic Kotlin will need to be able to “decode” complex Kotlin when needed.

To that end, this book is divided into two major sets of chapters: the core chapters, and the “WTF?” chapters.

The Core Chapters

The core chapters, starting with the next chapter, will focus on:

- Basic Kotlin syntax: creating classes, functions, properties, and expressions
- Widely-used Kotlin idioms: things that are a little strange with respect to other programming languages but get used a lot in Kotlin to reduce the size of the source code

If you master the concepts outlined in these chapters, you should be able to be a productive Kotlin developer.

The “WTF?” Chapters

The remaining chapters focus on advanced Kotlin syntax, for the sorts of things that most developers will not use very frequently. Most likely, you will encounter that syntax when reading existing Kotlin code. So these chapters are organized around specific bits of syntax that you might encounter, explaining their role and use.

A Few “Hello, World!” Examples

Let’s start looking at some basic “hello, world!” bits of Kotlin code, and along the way see how you can run this code in the Klassbook or elsewhere.

Just the Statement

```
println("hello, world!")
```

(from ["Hello, World! \(In a Function\)" in the Klassbook](#))

The above code snippet is a simple Kotlin statement, one that prints “hello, world!” somewhere.

A FEW “HELLO, WORLD!” EXAMPLES

Below the code snippet, you will see a link. If you are in an ebook reader that supports links, you can click it to visit the Klassbook page corresponding to that code snippet:

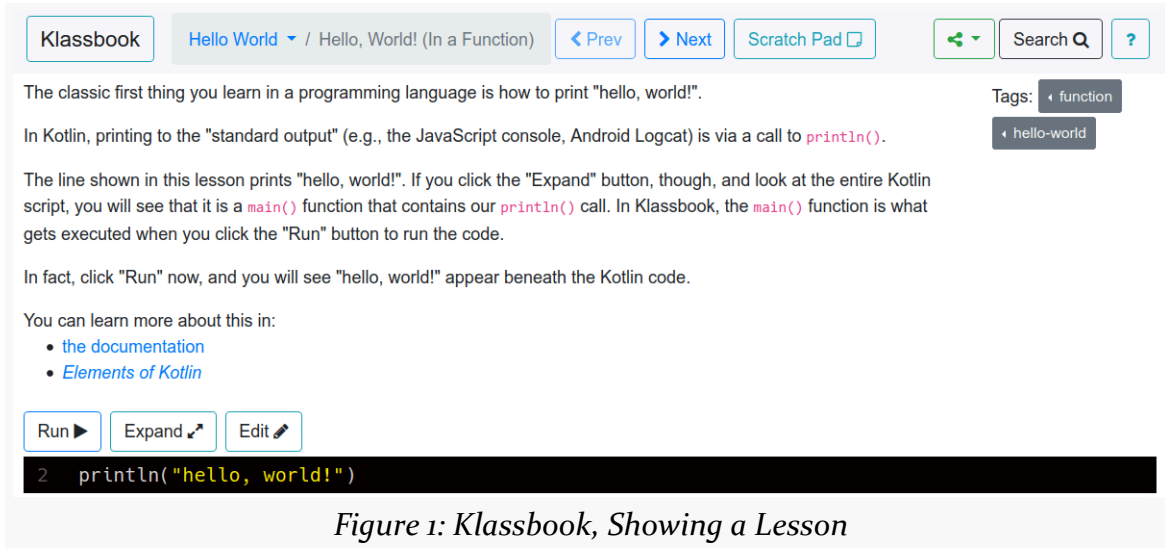


Figure 1: Klassbook, Showing a Lesson

If you click the “Run” button, the results of running the Kotlin code will appear below the code in the browser:



Figure 2: Klassbook, Showing the Results

Wrapped in a Function

In reality, though, there is a bit more to this Kotlin code than what you will initially see.

When you see an “Expand” button above the code, that means some of the code is being hidden from view, to focus your attention onto some specific set of lines.

A FEW “HELLO, WORLD!” EXAMPLES

Clicking the “Expand” button will show the entire bit of Kotlin that we are executing:

```
fun main() {  
    println("hello, world!")  
}
```

This `main()` function works similarly to the static `main()` method in a Java program, indicating the function that should be called to run the Java program.

Each of the Klassbook snippets has a `main()` function. Sometimes the full Kotlin snippet will be visible in the page, and there will be no “Expand” button. Other times, the `main()` function will be hidden, but the “Expand” button will show the full snippet, not just one portion.

Wrapped in a Class

The previous chapter mentioned that Kotlin is a class-based object-oriented language. So, if we wanted, we could wrap this code in a class and use that.

```
class HelloWorld {  
    fun speak() {  
        println("hello, world!")  
    }  
}  
  
fun main() {  
    HelloWorld().speak()  
}
```

(from ["Hello, World! \(In a Class\)"](#) in the Klassbook)

If you run this, you get the same output as the simple statement. We just wrapped it in a class to show off Kotlin classes a bit.

So, What Does This Do?

Let’s examine each of the pieces of what we have been executing, and use those to show you where we will be exploring in this book.

```
println("hello, world!")
```


A FEW “HELLO, WORLD!” EXAMPLES

First, we have "hello, world!". This is a Kotlin string. It looks much like strings in Java, Ruby, JavaScript, and many other programming languages. Kotlin strings are a bit more powerful than those in Java, as Kotlin adopted some of the string features offered in other programming languages. We will take a closer look at strings in [the next chapter](#).

Next, we have `println()`. This is a Kotlin function, one that accepts a string as input and prints it... somewhere. The exact location of where that string gets printed will depend a bit on where Kotlin is running:

- In a REPL, it will print wherever the REPL prints stuff, such as the Klassbook showing it below the code
- In an Android app, it will appear in Logcat
- In a command-line program, it will print to the terminal window or wherever “standard out” has been redirected to
- And so on

Java programmers will wonder where the class and method are. In Java, we do not have standalone lines of code. Rather, all code goes in a class somewhere, usually inside of a method in that class. Usually we import the class and then call the method on that class, such as `System.out.println()`, which is the Java equivalent of `println()` in Kotlin. Kotlin supports some statements and functions that are not part of any class, akin to how Ruby and JavaScript can have their own global methods and functions.

Eventually, we wrapped the `println()` in a function and put it inside a class:

```
class HelloWorld {  
    fun speak() {  
        println("hello, world!")  
    }  
}
```

This works similarly to how classes and methods are defined in Java and Ruby. JavaScript’s object model is somewhat different but, in the end, you can accomplish much the same thing. We will examine functions [in an upcoming chapter](#) and classes [a bit after that](#).

We then created an instance of our class and called our `speak()` function:

```
HelloWorld().speak()
```

A FEW “HELLO, WORLD!” EXAMPLES

Kotlin has very terse syntax for creating an instance of a class: just use the class name like a function. So, `HelloWorld()` returns an instance of a `HelloWorld` object. There is no need for a keyword, the way Java, Ruby, or JavaScript uses `new`.

Calling a function on an object then works much like other object-oriented languages — use `.` followed by the function call (`speak()`).

Sometimes, our functions take parameters, as we saw with `main()`:

```
fun main(args: Array<String>) {  
    HelloWorld().speak()  
}
```

Here, `args` is a parameter, of type `Array<String>`. Kotlin is a “strongly typed” language: everything in Kotlin is tied to some type. This is similar to Java but distinct from Ruby and JavaScript, which do not use types in this fashion. We will explore parameters and their use of types more in [the chapter on functions](#). We will also look at that angle-bracket syntax — part of Kotlin’s support for generics — [in an upcoming chapter](#).

Over the next several chapters, we will be examining more about these basic building blocks of Kotlin syntax, plus we will be experimenting with more elaborate code snippets.

Using the Klassbook

This book has hundreds of code snippets, most of which will have corresponding Klassbook pages. The companion book [Elements of Kotlin Coroutines](#) also contributes its snippets to the Klassbook as well.

The Klassbook has several additional features for helping you learn Kotlin.

Editing the Snippets

Each page has an “Edit” button above the code. Clicking that converts the code into a live editor, where you can make changes. It also expands the code, like the “Expand” button does, so you can work with the full snippet.

Once you have made changes, click “Run” to run the revised Kotlin code. It will take a few seconds for the Kotlin code to be “transpiled” into JavaScript and then run in your browser.

A FEW “HELLO, WORLD!” EXAMPLES

If you make a syntax error, the error messages will appear below the code, highlighted in red.

The Scratch Pad

Each page has a “Scratch Pad” button in the nav bar. Clicking that takes you to a page that just contains the code editor and “Run” button, without the rest of the content or widgets that you see on lesson pages. You can use the scratch pad for your own Kotlin experiments.

Note, though, that you are limited to 1KB of Kotlin source code in the editor. You can type in more than that, but anything longer will not run.

Navigating

You already saw how to go from this book to the Klassbook to bring up a relevant lesson. However, there are other navigation options in Klassbook itself.

Prev/Next Buttons

The nav bar has “Prev” and “Next” buttons to advance through the lessons in order. That will generally follow the order in which they appear in the book, though there will be occasional deviations.

Tags

Each lesson has one or more tags associated with it. Those appear on the side of the page. Clicking a tag will bring up a scrollable list of other lessons that have the same tag.

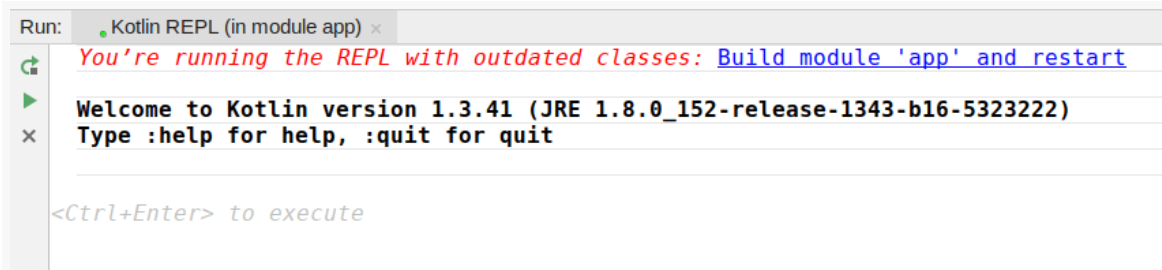
You can also click the “Search” button in the nav bar to bring up a list of all tags. Clicking on a tag there once again brings up a list of lessons associated with that tag.

Running These Snippets in an IDE

You are also welcome to try these Kotlin snippets outside of the Klassbook.

A FEW “HELLO, WORLD!” EXAMPLES

Android Studio and IntelliJ IDEA each has a Kotlin REPL, available from the Tools > Kotlin > Kotlin REPL main menu option:



```
Run: Kotlin REPL (in module app) x
You're running the REPL with outdated classes: Build module 'app' and restart
Welcome to Kotlin version 1.3.41 (JRE 1.8.0_152-release-1343-b16-5323222)
Type :help for help, :quit for quit
<Ctrl+Enter> to execute
```

Figure 3: Android Studio Kotlin REPL, As Initially Launched

The Klassbook is set up to run your `main()` function when you click the “Run” button. Android Studio’s and IDEA’s REPL is not. If you copy a Klassbook snippet and paste it into the REPL, you will need to call `main()` yourself, by simply typing `main()` onto a separate line after the pasted-in code. Then, click the green “run” triangle button, or press `Ctrl-Enter` on Windows/Linux machines, to run the code. The output will be printed in green italics immediately below your code:



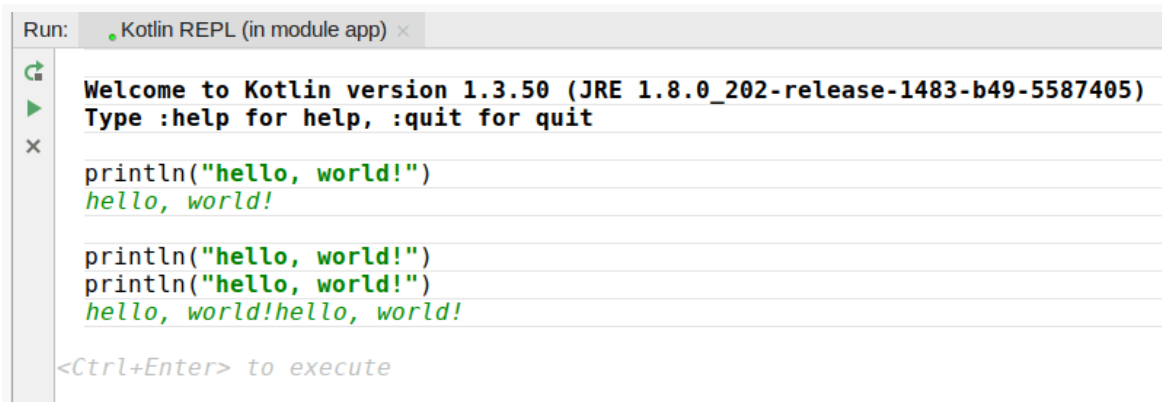
```
Run: Kotlin REPL (in module app) x
Type :help for help, :quit for quit
fun main() {
    println("hello, world!")
}
main()
hello, world!
<Ctrl+Enter> to execute
```

Figure 4: Android Studio Kotlin REPL, After Running Code

A few code snippets will not work in the Android Studio/IDEA REPL. That is because Klassbook is using Kotlin/JS, whereas Android Studio and IDEA use Kotlin/JVM. Most of the Kotlin that we see will be the same between those platforms, but there will be occasional differences.

A FEW “HELLO, WORLD!” EXAMPLES

Also, any code snippets that print more than one line will not work well in the IDE-based REPLs. They will tend to combine all the output onto one line:



```
Run: Kotlin REPL (in module app) x
Welcome to Kotlin version 1.3.50 (JRE 1.8.0_202-release-1483-b49-5587405)
Type :help for help, :quit for quit

println("hello, world!")
hello, world!

println("hello, world!")
println("hello, world!")
hello, world!hello, world!

<Ctrl+Enter> to execute
```

Figure 5: Android Studio Kotlin REPL, Showing Poor Results

Basic Types and Expressions

Pretty much every programming language has the concept of data types, things like integers and strings and so forth. Kotlin is no exception. Kotlin's basic types are modeled somewhat after Java, as Kotlin's original target environment was the JVM. However, there are some differences that you will encounter in your Kotlin development work.

Basic Types and “Objectness”

From Kotlin's standpoint, everything is an object. This stands in contrast to some languages, like Java, where basic types are “primitives” but have object counterparts. So, in Java, `int` is a primitive and `Integer` is the corresponding object type. Kotlin does away with that distinction, so everything is an object, including things like numbers.

Numbers

Kotlin supports the same basic numeric types as does Java. There are four integral types:

- Byte (8-bit representation)
- Short (16 bits)
- Int (32 bits)
- Long (64 bits)

And there are two floating-point types:

- Float (32 bits)

- Double (64 bits)

These are also fairly common across other programming languages, and so it should be fairly easy to get used to them.

Number Literals

Sometimes, you will use numbers directly in your code, as literal values, such as 1234 or 3.14159. By default, these will be an Int and a Double, respectively. If you want to make a literal be a Long, add an L suffix:

- 1234 is an Int
- 1234L is a Long

If you want a floating-point literal to be treated as a Float, not a Double, append f to the number.

If you run this snippet on the [Klassbook](#) site:

```
println(1234::class)
println(1234L::class)
println(3.14159::class)
println(3.14159f::class)
```

(from "[Printing the Class of Numbers](#)" in the [Klassbook](#))

...you should get:

```
class Int
class Long
class Double
class Float
```

In Android Studio or IntelliJ IDEA, you will get:

```
class kotlin.Int
class kotlin.Long
class kotlin.Double
class kotlin.Float
```

Here, the `::class` syntax says “give me a reference to the Kotlin class for this object”. So, `1234::class` returns an object that tells you what class 1234 is. In this case, it is `kotlin.Int` — Int being the class, and that class being in a “package” named

BASIC TYPES AND EXPRESSIONS

`kotlin`. We will talk more about packages [in an upcoming chapter](#), so just ignore that part for now.

For longer numbers, if you like, you can use underscores for the “thousands separator”. So this:

```
println(1_234)
```

(from ["Using Underscores for Thousands Separators" in the Klassbook](#))

gives you:

```
1234
```

Technically, those underscores can go anywhere, so `1_2_3_4` is also perfectly valid.

For integral types, the default representation is decimal format. You can define literals in hexadecimal by using an `0x` prefix (`0xFFA4C639`). You can also define binary literals by using an `0b` prefix (`0b10110100`).

(and for those of you wondering about octal support, like Java has... it is the 21st Century, and nobody uses octal anymore)

Mathematical Expressions

Your basic mathematical operators are available in Kotlin as they are in most other programming languages:

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `%` for the remainder after division (“modulo”)

Parentheses can be used for grouping to offer manual control over the order of operations. The default order of precedence puts multiplicative operations (`*`, `/`, `%`) higher than additive operations (`+`, `-`).

So:

```
println(1+2*3)
println((1+2)*3)
```


(from ["Basic Math" in the Klassbook](#))

results in:

```
7
9
```

Boolean

Kotlin offers a Boolean type, with two values: true and false. These work pretty much as they do in other programming languages. For example, we will see how to use Boolean values for if expressions [later in the book](#).

The typical sorts of Boolean operators that you may be used to from other programming languages are available to us:

- && for a logical AND
- || for a logical OR
- ! for a logical NOT

Strings

A String, representing a piece of text, is a commonplace type in most programming languages, and Kotlin is no exception. However, compared to languages like Java, Kotlin offers greater expressiveness when declaring string literals.

String Quoting Options

It is fairly likely that you are used to programming languages where strings are denoted by double-quote (") characters, such as in our “hello, world” example:

```
println("hello, world!")
```

(from ["Hello, World! \(In a Function\)" in the Klassbook](#))

In Java, that is your only option. Other languages, like Ruby, offer lots of possible ways to declare strings, such as single-quoted values ('Hello, world!'). Kotlin falls in between, offering two string quoting options: double-quotes and triple-quotes (""").

A double-quoted string works like its Java counterpart:

BASIC TYPES AND EXPRESSIONS

- It cannot contain newlines
- Special characters, such as newlines or tabs, need to be indicated via “escape sequences”

There are eight escape sequences supported in Kotlin:

- `\t` for tab
- `\b` for backspace
- `\n` for newline
- `\r` for “carriage return” (if you are under 50 years old, ask your grandparents what a “carriage” was with respect to a “typewriter”)
- `\'` for a single quote
- `\"` for a double quote
- `\$` for a dollar sign
- `\\` for a backslash

Anything else can be encoded using Unicode escape sequences (e.g., `\u221E` for the infinity symbol: ∞)

A triple-quoted “raw” string, on the other hand, does not support escape sequences but does support embedded newlines, tabs, and anything else. So, this Kotlin:

```
println("""Hello,  
world!""")
```

(from ["Raw Strings" in the Klassbook](#))

produces this output:

```
Hello,  
world!
```

This allows you to directly express strings that otherwise would be a mess of text and escape sequences, resulting in more readable code.

However, indents then become a problem. If your IDE wants to indent code to keep things aligned, it might add spaces or tabs inside of your string that you do not want. To help with this, you can use a `trimMargin()` function on `String` to eliminate unwanted indentation.

For example, let’s look at:

BASIC TYPES AND EXPRESSIONS

```
fun main() {
    println("""Hello,
world with extra whitespace!""")
    println("""Hello,
>world using >!""".trimMargin(">"))
    println("""Hello,
|world using |!""".trimMargin())
}
```

(from "[trimMargin\(\)](#)" in the *Klassbook*)

The first `println()` call results in extra whitespace being embedded in the result, as the second line is indented:

```
Hello,
  world with extra whitespace!
```

The other two `println()` calls use `trimMargin()`. The latter of those uses the default pipe (`|`) as the margin indicator, while the other uses a custom character (`>`), passing that character as a parameter to `trimMargin()`. Both remove our indent:

```
Hello,
world using >!
Hello,
world using |!
```

String Expressions

String concatenation works using the `+` operator as you would expect:

```
println("Hello, " + "world!")
```

(from "[String Concatenation](#)" in the *Klassbook*)

However, you will find a lot less string concatenation used in Kotlin than you may be used to in other languages like Java. That is because Kotlin, like Ruby, supports “string interpolation”, where expressions embedded in string literals can be evaluated directly. We will see that in action [later in the book](#).

Characters

Occasionally, we need to deal with individual characters, not longer strings. Characters can be represented by very short strings, if desired. However, just as Java has `char` and `Character` to represent individual characters, Kotlin too has a

Character class.

In Kotlin, as with Java, a Character literal is represented by a single-quoted character:

```
println('a')
```

(from ["Characters" in the Klassbook](#))

Also, similar to Java, escape sequences can be used for individual characters, much as they can with strings.

A Quick Note About Equality

In most programming languages, the == operator represents some form of an equality check. The exact meaning of == varies by language.

For example, in the case of Java, == means instance equality, where the left hand and right hand objects are the exact same object. For content equality, Java uses an equals() method. This has been known to confuse many developers over the years:

```
String value = "something"
System.out.println(value.toUpperCase().toLowerCase() == value); // prints false
System.out.println(value.toUpperCase().toLowerCase().equals(value)); // prints true
```

In Kotlin, == is used to mean content equality, the way it is for many other programming languages. === (three equals signs) is the operator for instance equality:

```
val value = "something"
println(value.toUpperCase().toLowerCase() == value) // prints true
println(value.toUpperCase().toLowerCase() === value) // prints false in Kotlin/JVM and true in Kotlin/JS
```

(from ["Content and Instance Equality" in the Klassbook](#))

Here, val defines a variable, named value. We will explore variables in greater detail [later in the book](#).

What you get as the result of a content equality check (==) is determined by Kotlin. What you get as the result of an instance equality check (===) is determined a bit by Kotlin but mostly by the Kotlin environment. For example, if you run the above code in the Kotlin/JS-based Klassbook, you get:

BASIC TYPES AND EXPRESSIONS

```
true  
true
```

If you run that same code in an Android app or some other Kotlin/JVM project, you get:

```
true  
false
```

The discrepancy has to do with the way Kotlin “transpiles” its code into JavaScript and the way that JavaScript compares strings, as is outlined in [this Stack Overflow answer](#).

For the vast majority of your Kotlin work, you will be using content equality, not instance equality, and therefore the difference in how string instance equality is handled will not affect you.

Variables... Whether They Vary or Not

A computer program *usually* needs something more than simple literals. We need to perform calculations, and sometimes we need to hold those results somewhere. Classically, in programming, we call such things “variables”. In Kotlin, that winds up being a bit of a curious name, as “variable” suggests a value that can vary... which is not true in all Kotlin cases.

Declaring Variables

We have to hold our data *somewhere*, and in some cases that “somewhere” is a variable. Different languages have different ways of declaring variables:

- Do you need a keyword, like `var`, or not?
- Do you need a data type, like `int`, or not?
- Do you need to provide an initializer to establish the variable’s value, or not?

In Kotlin, the required items are a keyword (`var` or `val`), the name, and (in most cases) an initializer. The type may or may not be required, depending on the initializer.

Typeless Declarations

If you are initializing the variable as part of declaring it, Kotlin will infer the type of the variable.

For example:

VARIABLES... WHETHER THEY VARY OR NOT

```
fun main() {  
    var count = 5  
  
    println(count::class)  
}
```

(from "[Variables](#)" in the *Klassbook*)

This gives us the same output as what we would get if we printed `5::class`:

- `class Int` in the *Klassbook* or other Kotlin/JS environments
- `class kotlin.Int` in Android or other Kotlin/JVM environments

Here, Kotlin sees that our variable initialization code evaluates to an `Int`, and so it declares the variable `count` as being of type `Int`.

This is one of many ways that Kotlin tries to keep the “ceremony” down and keep the language concise.

Typed Declarations

It is also possible to declare a variable with a type:

```
var count: Long = 5  
  
println(count::class)
```

(from "[Typed Variables](#)" in the *Klassbook*)

This says that the variable `count` is a `Long`. Even though `5` is an `Int` (`5L` would be the `Long` equivalent), Kotlin is kind to you and will convert the literal value to a `Long` as part of assigning it to the variable.

In many cases, the type is unnecessary, as Kotlin can default the variable’s type to be whatever the type is of our initial value. But there will be times when you want a variable to have a different type than the initial value, and we will see a few examples of that as we proceed through the book.

Declaring Read-Only Variables

A closely-related keyword to `var` is `val`. It too can declare a variable... except that `val` variables are read-only. After you initialize them, they cannot be changed.

VARIABLES... WHETHER THEY VARY OR NOT

So, this works:

```
val count = 5

println(count::class)
println(count)
```

(from "[Read-Only Variables](#)" in the *Klassbook*)

This prints the type and the value, such as this *Klassbook* output:

```
class Int
5
```

You can modify a var value:

```
var count = 5

println(count)

count = 7

println(count)
```

(from "[You Can Modify Normal Variables](#)" in the *Klassbook*)

This gives us:

```
5
7
```

However, this does not:

```
var count = 5

println(count)

count = 7

println(count)

val readOnly = 5

println(readOnly)
```


VARIABLES... WHETHER THEY VARY OR NOT

```
readOnly = 7  
  
println(readOnly)
```

(from ["You Cannot Modify Read-Only Variables" in the Klassbook](#))

If you try running it, you will get a compile error:

```
e: klassbook.kt: (14, 3): Val cannot be reassigned
```

While we can fill a value into a `var` whenever we want, we can only fill a value into a `val` once, usually where we declare it.

Prefer val Over var

It may seem that `var` is more useful than `val`. After all, a `val` can only hold one value, whereas we can reassign values to a `var` as needed.

Yet you will find that the vast majority of Kotlin code uses `val`. If anything, `var` tends to be considered as a “workaround”, for cases where `val` cannot be used for one reason or another.

We will hold off on a detailed explanation of this philosophy, and we will return to this point [later in the book](#). For the moment, take it on faith that we have good reasons to prefer `val` over `var`, and that is why this book will be using `val` much more than `var`.

String Interpolation

Many programming languages offer some form of “string interpolation”, where strings can contain programming language expressions directly in them. For example, you can do this in JavaScript, using backticks to enclose the string:

```
let count = 5;  
console.log(`The value of count is ${count}`);
```

This will print the following to the console:

```
The value of count is 5
```

Ruby has a similar capability for double-quoted strings:

VARIABLES... WHETHER THEY VARY OR NOT

```
count = 5
puts "The value of count is #{count}"
```

Kotlin's string interpolation syntax is reminiscent of JavaScript's. However, for a simple variable reference, you can skip the braces and just use \$:

```
val count = 5

println("The value of count is $count")
```

(from ["String Interpolation" in the Klassbook](#))

Arbitrary Kotlin expressions can be used with `${}` syntax:

```
val count = 3

println("The value of count is not ${count+2}")
```

(from ["String Interpolation with Expressions" in the Klassbook](#))

This can be very handy for assembling a string from many pieces, compared to using something like `StringBuilder` in classic Java.

More Operators

In the preceding chapter, we saw [mathematical expressions](#) and various operators, such as `+` and `-`. Those operators can work on literal values as we saw. Not surprisingly, they can also work with variables:

```
val count = 5
val more = count + 2

println(more)
```

(from ["Variables and Operators" in the Klassbook](#))

As you might expect, this prints 7 as the output.

However, now that we have variables, we have more operators that we can use!

Increments and Decrements

Akin to many other programming languages, we can use `++` and `--` operators to increment and decrement a variable by 1:

VARIABLES... WHETHER THEY VARY OR NOT

```
fun main() {
    var postIncrement = 5

    println("postIncrement ${postIncrement} ${postIncrement++} ${postIncrement}")

    var preIncrement = 5

    println("preIncrement ${preIncrement} ${++preIncrement} ${preIncrement}")

    var postDecrement = 5

    println("postDecrement ${postDecrement} ${postDecrement--} ${postDecrement}")

    var preDecrement = 5

    println("preDecrement ${preDecrement} ${--preDecrement} ${preDecrement}")
}
```

(from "[Increment and Decrement Operators](#)" in the *Klassbook*)

Here, we use string interpolation with `println()` to show the value of a variable before, “during”, and after an increment or decrement operation.

The results are:

```
postIncrement 5 5 6
preIncrement 5 6 6
postDecrement 5 5 4
preDecrement 5 4 4
```

So:

- Post-increment (`variable++`) increments the variable after the use
- Pre-increment (`++variable`) increments the variable before the use
- Post-decrement (`variable--`) decrements the variable after the use
- Pre-decrement (`--variable`) decrements the variable before the use

Of course, these only work with `var`, as the value of a `val` is “immutable” and cannot be changed.

Augmented Assignments

Similarly, there are also operators that evaluate a mathematical expression and assign it to the `var` in one shot, such as `+=`:

VARIABLES... WHETHER THEY VARY OR NOT

```
var count = 5  
  
count += 2  
  
println(count)
```

(from ["Augmented Assignments" in the Klassbook](#))

This prints 7, showing that count had its value incremented by 2. This is the equivalent of:

```
var count = 5  
  
count = count + 2  
  
println(count)
```

There are -=, *=, /=, and %= operators as well, combining those mathematical operators with assignments.

Unary Operators

Most programming languages offer ! (or some equivalent) as a “unary operator”, which inverts the value of a Boolean. Kotlin has that, along with a - negation operator that inverts the sign of a number:

```
val thisIsTrue = true  
  
println(!thisIsTrue)  
  
val whySoNegative = -5  
  
println(-whySoNegative)
```

(from ["Unary Operators" in the Klassbook](#))

As you might expect, this prints:

```
false  
5
```

No Automatic Number Conversions

In many programming languages, you can convert numbers between shorter

VARIABLES... WHETHER THEY VARY OR NOT

representations (like a Java `int`) and longer representations (like a Java `long`) just via assignments. The compiler knows to “upsized” the value.

In Kotlin, though, that only works for literals, not variables.

So, for example, as we saw earlier, this works:

```
var count: Long = 5

println(count::class)
```

(from ["Typed Variables" in the Klassbook](#))

However, this does not compile:

```
val thisIsInt = 5
val thisIsLong : Long = thisIsInt

println(thisIsLong::class)
```

The compiler error will be something akin to:

```
error: type mismatch: inferred type is Int but Long was expected
val thisIsLong : Long = thisIsInt
```

Kotlin wants you to intentionally perform such conversions. There is a `toLong()` function that does the trick:

```
val thisIsInt = 5
val thisIsLong : Long = thisIsInt.toLong()

println(thisIsLong::class)
```

(from ["Type Conversions" in the Klassbook](#))

Hey, What About ?

If you read through Kotlin code, you will see variables declared with questions for types:

```
var something : Boolean? = null

println("something was: $something")
```

VARIABLES... WHETHER THEY VARY OR NOT

```
something = true
println("something is now: $something")
```

Kotlin has many features that are not seen very often in other programming languages. One of the most important of these features is how Kotlin handles null values. Boolean is a type that is either true or false, while Boolean? is a type that is either true, false, or null.

We will explore “nullability” in much greater detail [later in the book](#). For now, take it on faith that we can assign null to a variable whose type ends in ?, and we cannot assign null to a variable whose type does not end in ?. So, where you see ? in types, watch out for null values.

Functions

We saw our first Kotlin function [several chapters ago](#):

```
println("hello, world!")
```

(from ["Hello, World! \(In a Function\)" in the Klassbook](#))

Now, let's explore functions in greater detail, as they offer a number of features that are not quite as common among the peer set of languages that we are comparing Kotlin to (Java, Ruby, JavaScript).

Functions with Parameters

Our `main()` functions have had no parameters. However, functions can take parameters, and most functions that you will write will wind up with one or more parameters:

```
fun main() {
    lessTrivial(1, 1)
}

fun lessTrivial(left: Int, right: Int) {
    println(left + right)
}
```

(from ["Functions with Parameters" in the Klassbook](#))

In the function declaration, a parameter is a name, followed by a colon, followed by a type. The function declaration can have no parameters, a single parameter, or a comma-delimited list of parameters. So, here, we have two parameters, named `left` and `right`, that are both of type `Int`.

The default approach for calling such a function is to simply provide values for those parameters, in the same order as those parameters appear in the function declaration. That is what we are doing with our `lessTrivial(1, 1)` call. However, as we will see [later in this chapter](#), there are alternative ways of calling this function.

Functions with Return Types

The functions shown so far are Kotlin's equivalent of Java methods that return void, in that they are not returning any values to the caller.

(technically, they are returning `Unit`, which we will explore [later in the book](#))

However, as with functions and methods in most programming languages, Kotlin functions can return values, using the `return` keyword:

```
fun main() {
    println(simpleReturn(1, 1))
}

fun simpleReturn(left: Int, right: Int) : Int {
    return left + right
}
```

(from "[Functions with Return Types](#)" in the *Klassbook*)

However, to be able to return a value, the function needs to declare the type of what is being returned. In many languages, such as Java, that “return type” is declared at the beginning of the function declaration. In Kotlin, it is declared at the end, after the closing parenthesis, separated by a colon. So, here, we are returning an `Int`, by virtue of that `: Int` after the `fun simpleReturn(left: Int, right: Int)` part.

Code that calls the function then can use that returned value. In the above code snippet, we are printing it to standard output. You could also store it in a variable:

```
fun main() {
    val result = simpleReturn(1, 1)

    println(result)
}

fun simpleReturn(left: Int, right: Int) : Int {
    return left + right
}
```

(from ["Assigning Function Results" in the Klassbook](#))

Local Variables

Variables declared inside of a function are considered to be local variables. They are available inside of the function but not outside of it. And those variables go “out of scope” once the function returns.

So, we could rework our function to use a local variable to hold onto the calculation:

```
fun main() {
    println(actLocally(1, 1))
}

fun actLocally(left: Int, right: Int) : Int {
    val result = left + right

    return result
}
```

(from ["Local Variables" in the Klassbook](#))

Fancier Functions

Those basics are enough to allow you to write Kotlin code. However, there are a few additional features that are fairly prevalent in Kotlin programming. How much you use these features is up to you, but you will encounter them frequently and so it is important to understand how they work.

Expression Bodies

Frequently, our function bodies are lines of code wrapped in braces, as shown above.

Sometimes, you will encounter functions that are declared using an = instead:

```
fun main() {
    println(expressionBody(1, 1))
}

fun expressionBody(left: Int, right: Int) = left + right
```

(from ["Expression Bodies" in the Klassbook](#))

This is called “expression body” syntax.

It is designed to simplify cases where the entire method implementation is a single expression, such as adding two numbers together. Here, we replace a pair of braces and a return keyword with an `=`, which makes this code less verbose.

Expression Bodies and Types

Another thing that we eliminated in the above example is the return type. We did not need to declare that `expressionBody()` returns an `Int`, as the compiler can determine that on its own. It knows what the type of `left` is, and it knows what the type of `right` is. It even knows what the type of `left + right` is. So, it knows the type of `expressionBody()` as a result.

However, sometimes, you will need to override that type. For example, there will be cases where the expression evaluates to one type, but you want the function to return a supertype.

In those cases, you just add back the `:` syntax for declaring the return type:

```
fun main() {
    println(expressionBody(1, 1))
}

fun expressionBody(left: Int, right: Int) : Number = left + right
```

(from ["Expression Bodies and Return Types" in the Klassbook](#))

Here, we declare `expressionBody()` to return a `Number`. As it turns out, `Int` inherits from `Number`.

This particular example is silly. However, once we start getting into [classes](#) and inheritance, this capability becomes more important.

Why Bother?

In the end, this may not seem like a “big win”, for two reasons:

1. We are only saving a few characters
2. Not that many functions would appear to be simple enough that their entire implementation can be a single expression

FUNCTIONS

The first argument is true: we are not saving much on a per-function basis. However, these simplifications add up over time. A lot of focus in Kotlin is on offering these small simplifications, in a lot of places, which combine to make Kotlin code much more concise than the equivalent code in Java and other languages.

In terms of the second argument... it turns out that a lot more things in Kotlin can be written in terms of expressions than you might think. We will see more about that [in an upcoming chapter](#).

Default Parameter Values

Some languages offer default parameter values, such as JavaScript:

```
function increment(base, amount = 1) {  
  // something yummy  
}
```

...and Ruby:

```
def increment(base, amount = 1)  
  # something yummy  
end
```

Kotlin also offers default parameter values:

```
fun main() {  
  println(increment(1))  
}  
  
fun increment(base: Int, amount: Int = 1) = base + amount
```

(from ["Default Parameter Values" in the Klassbook](#))

Here, we can call `increment()` with either one or two values. If we only supply one value, the default value of 1 will be used for `amount`. If we supply two values, then the caller controls both `base` and `amount`.

The equivalent Java code requires two methods:

```
int increment(int base, int amount) {  
  return base + amount;  
}
```

FUNCTIONS

```
int increment(int base) {  
    return increment(base, 1);  
}
```

Named Parameters

Functions with lots of default values are often called using named parameters. In this form of a call, rather than identifying parameters by the order in which they appear in the call, you explicitly state the name of the parameter in the call itself. You can even “mix and match” positional parameters and named parameters:

```
fun main() {  
    println(increment(base = 1))  
    println(increment(amount = 10, base = 1))  
    println(increment(1, amount = 10))  
}  
  
fun increment(base: Int, amount: Int = 1) = base + amount
```

(from ["Named Parameter Values" in the Klassbook](#))

Here, in the first two `println()` calls, we specifically state both the name and the value of the parameters in the function call. In the third `println()` call, the first parameter is a traditional “positional” parameter, while the other one is named.

In this particular example, we are not gaining much from this syntax. However, complex functions with lots of default parameter values can gain a lot more from this. Suppose that we had a function that looked like this:

```
fun iCanHazCookie(name: String,  
    value: String,  
    maxAge: Long? = null,  
    expires: LocalDateTime? = null,  
    domain: String? = null,  
    path: String? = null,  
    secure: Boolean = false,  
    httpOnly: Boolean = false) : String {  
    // code to generate an HTTP Set-cookie header goes here  
}
```

(as a reminder, types with a `?` indicate parameters that support null values — we will explore this more [later in the book](#))

Here, six of the parameters have default values defined, and so they are optional

FUNCTIONS

when we call `iCanHazCookie()`.

Suppose that I want to create a cookie with a particular path, but I am willing to accept the defaults for the remaining five optional parameters. If we were stuck with positional parameters, we would have to call it like this:

```
val cookieHeader = iCanHazCookie("foo", "bar", null, null, null, "/")
```

In effect, the first three optional parameters are no longer optional, because we need them as placeholders.

With named parameters, this becomes simpler:

```
val cookieHeader = iCanHazCookie(name = "foo", value = "bar", path = "/")
```

...or even:

```
val cookieHeader = iCanHazCookie("foo", "bar", path = "/")
```

We can more easily skip over the parameters that we wish to ignore and accept the default values, because we can name the parameters that we *are* supplying, rather than relying on positions to identify them.

Traditionally, named parameters had to appear after all positional parameters. [Kotlin 1.4 relaxed that restriction somewhat](#). You can use names for any parameters, but up through the last positional parameter, the positions (not the names) are what matter.

Collections and Lambdas

Often in programming, we need to deal with collections of stuff. If you have worked with any major programming language, you will have dealt with collection types, such as:

- arrays or lists
- sets (collections of distinct unequal objects)
- maps (collections of key/value pairs)

Kotlin has these as well. Learning how to use them not only involves dealing with the actual types, but also with how to perform common operations on them, such as iterating over their contents.

Major Collection Types and Creation Functions

Kotlin has the aforementioned roster of collection types, and they form the most common collection types that you will use.

Kotlin/JVM also has access to other standard Java collection types, like `LinkedHashMap`, which we will discuss [later in this chapter](#). However, usually you will use those only if there is no native Kotlin equivalent.

Arrays

Kotlin has an `Array` class that is analogous to Java arrays. If you use Kotlin arrays, and you are running on Kotlin/JVM, your Kotlin arrays will be mapped to Java arrays.

Overall, arrays are not quite as popular in Kotlin as are [lists](#). However, you are welcome to use them, and they may be necessary if your code is going to

[interoperate with Java code.](#)

To create an array, the simplest approach is to use the `arrayOf()` utility function:

```
val things = arrayOf("foo", "bar", "goo")
println(things::class)
println(things)
```

(from ["Arrays" in the Klassbook](#))

Running this in the Klassbook gives us:

```
class Array
foo,bar,goo
```

Typed Arrays

Kotlin also has dedicated classes for arrays of primitive types:

- `BooleanArray`
- `ByteArray`
- `CharArray`
- `DoubleArray`
- `FloatArray`
- `IntArray`
- `LongArray`
- `ShortArray`

They have corresponding utility functions to create instances, such as `intArrayOf()`:

```
val things = intArrayOf(1, 3, 3, 7)
println(things::class)
println(things)
```

(from ["Typed Arrays" in the Klassbook](#))

```
class IntArray
1,3,3,7
```

As with `Array`, these type-specific arrays are not especially popular, except where you might need them for working with Java code that expects those primitive array types.

Lists

The basic one-dimensional collection type in Kotlin is the list. This is analogous to a Java `ArrayList`.

You can create instances of a list via `listOf()`, much like you create arrays with `arrayOf()` and the others mentioned above:

```
val things = listOf("foo", "bar", "goo")

println(things::class)
println(things)
```

(from ["Lists" in the Klassbook](#))

Technically, `List` is an [interface](#). The concrete implementation that we see will be an `ArrayList`:

```
class ArrayList
[foo, bar, goo]
```

An unusual aspect of lists is that `listOf()` creates an “immutable” list, one where we cannot replace items in the list. We will explore immutability more [later in this chapter](#).

Sets

Many programming languages offer a “set” collection type, which is a one-dimensional collection in which only distinct objects will be included. For example, if you try adding the four numbers 1, 3, 3, 7 to a list, you would have four entries, but adding them to a set results in a three-element set holding 1, 3, 7. The duplicate 3 value is ignored.

Kotlin has first-class support for sets, created using `setOf()`:

```
val things = setOf(1, 3, 3, 7)

println(things::class)
println(things)
println(things.size)
```

(from ["Sets" in the Klassbook](#))

As with `List`, `Set` is an interface. The concrete implementation that you will get

from `setOf()` is a `LinkedHashSet`:

```
class LinkedHashSet
[1, 3, 7]
3
```

Among other stuff, this code snippet prints out the size of things. `size` is a property of a `Set` (and of `List`, for that matter), providing the number of elements in the collection. Here we see that the duplicate 3 values were replaced by a single 3 in the `Set`.

And, as with lists, `setOf()` creates an immutable `Set`, which we will cover [later in this chapter](#).

Maps

Most programming languages have some support for a “map” or “dictionary” type, representing a key-value store. You can then place data into the collection as key-value pairs, to be able to eventually retrieve values given their keys. In Kotlin, we have a `Map` type that fills this role.

To create a hard-coded map — the way we have created hard-coded lists and sets — you can use the `mapOf()` utility function. This takes a comma-delimited list of key-value pairs, where the key and value are separated by the `to` keyword:

```
val things = mapOf("key" to "value", "other-key" to "other-value")

println(things::class)
println(things)
println(things.size)
```

(from "[Maps](#)" in the *Klassbook*)

Of course, you can combine these constructs, such as having a map of lists:

```
val things = mapOf("odd" to listOf(1, 3, 5, 7, 9), "even" to listOf(2, 4, 6, 8))

println(things::class)
println(things)
println(things.size)
```

(from "[Combining Collections](#)" in the *Klassbook*)

(in truth, `to` is actually a function, leveraging some special syntax that we will see [much later in the book](#))

Basic Usage

Creating collections is nice, but eventually, we need to get data in and out of them and otherwise manipulate them.

[] Syntax

To retrieve values from a collection (other than a Set), you can use []. For arrays and lists, the value passed into the [] operator is the 0-based index into the collection. For maps, the value passed into the [] operator is the key for which you wish to look up the corresponding value.

```
val thingsArray = arrayOf("foo", "bar", "goo")
println(thingsArray[0])

val thingsIntArray = intArrayOf(1, 3, 3, 7)
println(thingsIntArray[1])

val thingsList = listOf("foo", "bar", "goo")
println(thingsList[2])

val thingsMap = mapOf("key" to "value", "other-key" to "other-value")
println(thingsMap["other-key"])
```

(from ["Accessing Elements via \[\]" in the Klassbook](#))

Running this in the Klassbook will give you:

```
foo
3
goo
other-value
```

Things get a bit interesting if we start playing with that map of the lists of odd and even single-digit integers:

```
val oddEven = mapOf("odd" to listOf(1, 3, 5, 7, 9), "even" to listOf(2, 4, 6, 8))
println(oddEven["odd"].size)
```

This will fail to run, with a compile error:

```
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type List<Int>?
```

A Map may return null for a given key, as there may not be a value for that key. As it turns out, our Map has a value for the odd key, and so at runtime, this would succeed. And in most programming languages, we could actually compile and run this sort of operation. This fails in Kotlin, because of how Kotlin handles the possibility of null as a value. We will explore this in much greater detail [later in the book](#).

Typical Stuff

If you are used to working with collections in other programming languages, it will come as little surprise that Kotlin has a lot of the same basic functions for working with its collections.

In addition to the size property, you will have things like:

- `isEmpty()`, for a quick Boolean value based on whether the collection is empty
- `contains()`, which will return `true` if the collection contains the requested item (for maps, this checks the keys — use `containsValue()` to check the values)

For lists and arrays, you have things like:

- `first()`, to get the first element
- `last()`, to get the last element
- `firstOrNull()`, to get the first element or `null` if the collection is empty
- And so on

Conversion Functions

The one-dimensional collection types (arrays, lists, sets) have a rich set of conversion functions to migrate between types:

- Array offers `toList()` and `toSet()`
- List offers `toSet()` and `toTypedArray()`
- Set offers `toList()` and `toTypedArray()`

Immutability and Collections

Array objects that we create with `arrayOf()` are mutable. Not only can we use `[]` syntax to retrieve a value based on its index, but we can use `[]=` syntax to replace a value based on its index:

```
val thingsArray = arrayOf("foo", "bar", "goo")
thingsArray[1] = "something completely different"
println(thingsArray[1])
```

(from "[Array Modification](#)" in the *Klassbook*)

However, this fails if you try it with a `List` created using `listOf()`:

```
val thingsList = listOf("foo", "bar", "goo")
thingsList[1] = "something completely different"
println(thingsList[1])
```

You get some very strange syntax error messages:

```
e: klassbook.kt: (4, 1): Unresolved reference. None of the following candidates is applicable because of receiver type mismatch:
@InlineOnly public inline operator fun MutableMap.set(key: Int, value: String): Unit defined in kotlin.collections
e: klassbook.kt: (4, 11): No set method providing array access
```

The error is raised because `listOf()` returns a `List`, which does not have support for the `[]=` operator. A `List` is immutable: you cannot replace its members with other objects.

If you need a list where you can replace its members, you can use `mutableListOf()`:

```
val thingsList = mutableListOf("foo", "bar", "goo")
thingsList[1] = "something completely different"
println(thingsList[1])
```

(from "[Mutable Lists](#)" in the *Klassbook*)

Here, `thingsList` is actually a `MutableList`, which is a sub-type of `List` that

supports []=.

The same distinction holds true for Map. A Map is immutable, but a MutableMap created via `mutableMapOf()` can be modified.

We will explore more about Kotlin's philosophy towards immutability [in a later chapter](#). In general, aim to use immutable lists and maps, unless you have a very specific reason to do otherwise.

Introducing Lambda Expressions

One thing that you will use a *lot* in Kotlin development is the lambda expression. It is particularly important when working with collections, as lambda expressions are critical for performing common operations, such as:

- Looping over the elements
- Finding or filtering elements that match certain criteria
- Providing comparison logic for sorting elements
- And so on

What is a Lambda Expression?

A simple way to start thinking about lambda expressions is to consider them just as a set of Kotlin statements wrapped in curly braces:

```
val lambda = { println("Hello, world!") }
```

However, as the `val` indicates, a lambda expression itself is an object. You can hold onto them in variables, pass them as function parameters, and do everything else that you might do with other types of objects.

Kotlin lambda expressions closely resemble their Java counterparts, along with similar constructs in other languages, such as Ruby blocks.

If you receive a lambda expression, and you want to run it, call `invoke()`:

```
val lambda = { println("Hello, world!") }  
  
lambda.invoke()
```

(from "[Lambda Expressions](#)" in the *Klassbook*)

This prints “Hello, world!”, just as if you had just directly executed the `println()` call.

Where Do We Use Lambda Expressions?

At minimum, lambda expressions replace a lot of the “listener” or “callback” patterns that you might have used in Java. For example, rather than creating an implementation of a `Comparator` to support a `sort()` method, as we do in Java, you would use a lambda expression.

In particular, anything in Java where you might have used a listener or callback consisting of a class with a single method, a lambda expression will be the likely replacement.

Kotlin also lends itself towards the [functional programming](#) paradigm, where business logic is made up of linked function calls. We will explore that more when we look at [immutability](#).

But, in a nutshell: you will use lambda expressions a *lot* in Kotlin.

Lambda Expressions, Parameters, and Return Values

Like functions, lambda expressions can take parameters:

```
val squarifier = { x: Int -> x * x }  
  
println(squarifier.invoke(3))
```

(from ["Lambda Expressions and Parameters" in the Klassbook](#))

Here, `squarifier` takes one parameter, named `x`, whose type is `Int`. The `x: Int` syntax is the same as function parameters. The list of parameters to a lambda expression come at the beginning, with the “body” of the lambda expression following the `->`.

However, single-parameter lambdas are very common. For that, Kotlin offers a shorthand syntax: `it`. If the Kotlin compiler can determine the data type for the parameter, you can skip the parameter declaration and just refer to the parameter as `it`. We will see examples of this coming up shortly.

A lambda expression will return whatever its last statement does. In the case of `squarifier`, it has only one statement: `x * x`. So, that value is what the lambda

expression returns from the call to `invoke()`. So, our REPL console output is 9, because the lambda expression takes 3 as input, squares it, and returns that squared value.

Common Collection Operations

With lambda expressions in mind, let's look at some fairly common functions available on our collection classes, most of which use a lambda expression.

Iteration

A common thing to do with a collection is to iterate over its contents. There are two ways to do that in Kotlin: `forEach()` and `for`.

`forEach()`

The more common approach is to use the `forEach()` function:

```
val things = listOf("foo", "bar", "goo")
things.forEach { println(it) }
```

(from "[forEach\(\)](#)" in the *Klassbook*)

This results in:

```
foo
bar
goo
```

In other words, `forEach()` iterates over each member of the collection and invokes the lambda expression for each one in turn. Our lambda expression happens to print out the value of the lambda expression's parameter. In our case, we use the shorthand `it` reference to that parameter. Alternatively, we could have given it a custom name:

```
val things = listOf("foo", "bar", "goo")
things.forEach { thing -> println(thing) }
```

`forEach()` is a function, available for `List` and the other collection types. It may not *look* like a function, given that it lacks parentheses after the `forEach` name.

`forEach()` takes a lambda expression as a parameter. It may not *look* like a parameter, again due to the lack of parentheses.

In this case, this is a bit of simplified Kotlin syntax. A function that takes one lambda expression parameter can be called without the parentheses, if you are providing a literal lambda expression. So, the Kotlin shown above is equivalent to:

```
val things = listOf("foo", "bar", "goo")

things.forEach( { println(it) } )
```

(from "[forEach\(\) is a Function](#)" in the *Klassbook*)

That looks more like a conventional function call, albeit one that takes a lambda expression as a parameter. However, you will rarely see such calls written that way — the syntax that skips the unnecessary parentheses is far more common, if you are using a literal lambda expression.

The lambda expression takes a single parameter. The type of the parameter is determined by the definition of `forEach()` — just as a function can declare a parameter as being an `Int` or a `String`, it can declare a parameter as being a lambda expression that itself takes a certain parameter. We will see how to declare such functions [later in the book](#). For now, take it on faith that Kotlin can figure out what the data type of the parameter to the lambda expression is... so we do not need to declare that parameter manually ourselves. Instead, we use `it` to refer to that parameter, in this case passing it to `println()`.

Many functions on `List`, `Set`, etc. work this way: they take a single lambda expression as a parameter, where the lambda expression itself takes a single parameter, and we write the call without parentheses and by using `it` to refer to the lambda expression parameter.

Good Old-Fashioned `for`

`forEach()` is not the only way to iterate over the members of a collection. There is also `for`, working very similarly to `for` loops in other programming languages:

```
val things = listOf("foo", "bar", "goo")

for (thing in things) {
    println(thing)
}
```

(from ["for Loop" in the Klassbook](#))

The parenthetical expression after the `for` keyword consists of a name to use for an individual member (`thing`) and the collection (`things`), separated by the `in` keyword. The block of code that follows can then refer to an individual member by whatever name you gave it, and that block will be executed once for each member in sequence.

(technically, this block of code is not a lambda expression, but it is fairly similar)

`forEach()` is somewhat more popular, as it can be used in chained expressions, as we will see in upcoming sections. However, you can use whichever of them makes you more comfortable.

filter()

`filter()` applies a filtering lambda expression to a collection, returning a *new* collection that contains the subset of items from the original collection. If the lambda expression returns `true` for an item, that item is included in the result collection, otherwise it is not.

```
val things = listOf("foo", "bar", "goo")
things.filter { it[1]=='o' }.forEach { println(it) }
```

(from ["filter\(\)" in the Klassbook](#))

Here we have a chained expression. We `filter()` the `things` collection, then call `forEach()` on the output of `filter()`. While this can be written on one line, you will see it more often written with each expression starting on its own line:

```
val things = listOf("foo", "bar", "goo")
things
  .filter { it[1]=='o' }
  .forEach { println(it) }
```

`filter()` takes a lambda expression that accepts an item from the collection and returns a `Boolean`, where `true` means that the item should be accepted and passed along, while `false` means that the item should be rejected. So, running this code snippet in the Klassbook yields:

```
foo  
goo
```

...as bar does not have an o as the second character.

map()

`filter()` passes along a strict subset of the items in the collection, but the items themselves are unchanged.

`map()`, by contrast, passes along each item from the collection, transformed by a lambda expression that you supply:

```
val things = listOf("foo", "bar", "goo")  
  
things  
  .map { it.toUpperCase() }  
  .forEach { println(it) }
```

(from "[map\(\)](#)" in the *Klassbook*)

Here, we `map()` a `String` to another `String`, converting it to uppercase:

```
FOO  
BAR  
GOO
```

There is no requirement for `map()` to return the same type as it starts with, though:

```
val oneAndPrimes = listOf(1, 2, 3, 5, 7)  
  
oneAndPrimes  
  .map { 1.0 / it }  
  .forEach { println(it) }
```

(from "[map\(\) Type Conversions](#)" in the *Klassbook*)

This code snippet iterates over single-digit prime values and prints their reciprocal (1 divided by the number). Here, we use `1.0` to force a floating-point representation, and so we get `Double` results:

```
1.0
0.5
0.3333333333333333
0.2
0.14285714285714285
```

There are countless more of these sorts of functions, and we will see many of them as we proceed through the book. We will also explore this sort of “functional programming” approach [in an upcoming chapter](#).

So, whereas `filter()` takes a collection and returns a subset of its items, `map()` takes a collection and returns a transformed representation of its items.

Varargs

Many programming languages have the concept of “varargs”: parameters declared for a method or function that represent zero, one, or several values.

In Java, we use `...` syntax:

```
void something(String... lots) {
    if (lots.size>0) {
        // do something
    }
}
```

`lots` is treated as a Java array, in this case an array of `String` objects.

JavaScript just dumps everything not matched by named parameters into an arguments array:

```
function something() {
    if (arguments.length>0) {
        // do something
    }
}
```

Ruby uses `*` syntax:

```
def something(*lots)
    if lots.length > 2
        # do something
    end
end
```

Kotlin dedicates the `vararg` keyword to this role:

```
fun main() {
    reciprocate(1, 2, 3, 5, 7)
}

fun reciprocate(vararg values: Int) {
    values
        .map { 1.0 / it }
        .forEach { println(it) }
}
```

(from "[Varargs](#)" in the *Klassbook*)

Here, `reciprocate()` can accept zero parameters, one parameter, or many parameters. We assign the `vararg` a name (`values`), and the parameters are given to us in the form of an `Array`. Here, we use the same code as in the previous section to compute the reciprocal of each of those values and print them to the console.

What You Can Pass In

Typically, a `vararg` parameter has values passed in via a comma-delimited list (e.g., `reciprocate(1, 2, 3, 5, 7)`). Those do not have to be constants, though:

```
fun main() {
    val foo = 1
    val bar = 2
    val goo = 3
    val baz = 5
    val heyWhatComesAfterBaz = 7

    reciprocate(foo, bar, goo, baz, heyWhatComesAfterBaz)
}

fun reciprocate(vararg values: Int) {
    values
        .map { 1.0 / it }
        .forEach { println(it) }
}
```

(from "[Values for Varargs](#)" in the *Klassbook*)

Any expression, including simple variable references, can be supplied for values.

In addition, Kotlin offers special support for an existing `Array`. If you already have an

Array with the values that you want to supply, but the function that you are calling uses `vararg` (instead of an Array parameter), you can use the “spread operator”:

```
fun main() {
    val things = arrayOf("foo", "bar", "goo")

    capped(*things).forEach { println(it) }
}

fun capped(vararg strings: String) = strings.map { it.toUpperCase() }
```

(from "[Spread Operator](#)" in the *Klassbook*)

Here, our `capped()` function takes a `vararg` of `String` values and returns them capitalized. We have an Array of `String` values in the form of `things`. We can pass `things` to `capped()` by prefixing `things` with `*`. This basically says “take the contents of this array and add them each as individual parameters to the function call”.

Since we happened to write `capped()`, it would be simpler to just have `capped` take an Array parameter. If `capped()` were written by somebody else, though, and they chose to use `vararg`, then the spread operator still lets us use our Array.

Things get a bit strange with types that map to primitives, though. `vararg` expects the dedicated array types to be used with the spread operator. So, in the following example, the `reciprocate()` function takes a `vararg` of `Int`. For the spread operator to work, we need to use an `IntArray`, not a regular Array:

```
fun main() {
    val primes = intArrayOf(1, 2, 3, 5, 7)

    reciprocate(*primes)
}

fun reciprocate(vararg values: Int) {
    values
        .map { 1.0 / it }
        .forEach { println(it) }
}
```

(from "[Spread Operator with Primitive Types](#)" in the *Klassbook*)

Other JVM Collections

If you are using Kotlin/JVM — for example, if you are using Kotlin for ordinary

Android app development — you are welcome to use other JVM collection types, such as `LinkedList`. On the whole, you should stick with Kotlin collection types where possible, as the Kotlin types are more “natural” for use in Kotlin. But sometimes you will find a Java class that fits your scenario better, such as using a `LinkedHashMap` to implement an LRU-style cache.

Coming up in [a later chapter](#), we will see how to create instances of arbitrary classes — such as `LinkedList` — and how to work with them.

`get()` and `[]` Syntax

Earlier, we explored `[]` syntax for accessing elements in a `List`, where `[]` would wrap a 0-based index of the item in the list to retrieve.

We also explored using `[]` for accessing entries in a `Map`, where `[]` would wrap the key for which we want to try to retrieve a value.

In reality, `[]` syntax is available for *anything* that offers a one-parameter `get()` function. When Kotlin encounters `[]` syntax, it simply replaces that with the corresponding `get()` call.

So, for example, we also saw using `[]` syntax to get the Nth character from a `String`. `String` has a `get()` function, that takes a 0-based index and returns that character from the string. When we used `[]` syntax (`it[1] == 'o'`), Kotlin called `get()` on the `String`, passing in our index.

If, When, and While

Branching and looping are common operations in most programming languages. Most languages have some concept of `if` (branch based on a condition) and `while` (loop based on a condition). Kotlin has its takes on those control structures, plus a `when` structure that is a bit reminiscent of things like Java's `switch` structure.

So, let's take a look at `if`, `when`, and `while`, and see how Kotlin handles them the same as — and in some cases, differently than — other languages.

If

`if` is nearly ubiquitous in computer programming, whether that is Java:

```
if (i>10) {  
    // do something  
}  
else {  
    // do something else  
}
```

...JavaScript:

```
if (i>10) {  
    // do something  
}  
else {  
    // do something else  
}  
  
// yes, it is the same as the Java syntax
```

IF, WHEN, AND WHILE

...or Ruby:

```
if (i>10)
  # do something
else
  # do something else
end
```

On the surface, a Kotlin `if` works the same way:

```
val i = 3

if (i > 10) {
    println("something")
}
else {
    println("something else")
}
```

(from "[if](#)" in the *Klassbook*)

As with most languages, the `else` clause is optional, if you do not have anything special that you want to do in that situation:

```
val i = 3

if (i > 10) {
    println("something")
}

println("the if is done")
```

(from "[if Without else](#)" in the *Klassbook*)

For single-line conditional work, you can skip the braces:

```
val i = 3

if (i>10) println("something") else println("something else")
```

(from "[Single-Line if](#)" in the *Klassbook*)

`if` also supports `else if` in addition to a simple `else`:

```
val i = 3

if (i > 10) {
```

IF, WHEN, AND WHILE

```
println("something")
}
else if (i > 2) {
    println("something else")
}
else {
    println("and now for something completely different")
}
```

(from "[else if](#)" in the *Klassbook*)

However, as you will see in the next section, the convention in Kotlin development is to use `when` instead of `if` for such scenarios.

When

`when` is distinctive. `when` is powerful. `when` is used a *lot* in Kotlin programming. And `when` is somewhat different than what you'll see in many other programming languages.

What We Do Elsewhere

The closest analogy to `when` in Java is the `switch` statement:

```
switch(foo) {
    case 1:
        // do something
        break;
    case 2:
        // do something else
        break;
    default:
        // like, whatever
}
```

Here, depending on the value of `foo`, we evaluate one set of statements that follows the matching case (or the default statements, if there are no matches).

JavaScript also has a `switch`, with the same syntax as Java.

Ruby uses `case`, and it is the closest of the three in terms of matching the power of `when`:

IF, WHEN, AND WHILE

```
case foo
  when 1
    # do something
  when 2
    # do something else
  else
    # like, whatever
end
```

The Basic Use of when

The most commonly-seen form of when looks like a Java/JavaScript switch or a Ruby case, where you supply a value for comparison and set up different branch conditions to test against that value:

```
val i = 3

when (i) {
  1 -> println("something")
  2 -> println("something else")
  else -> println("and now for something completely different")
}
```

(from "[when](#)" in the *Klassbook*)

Each branch has a value (1 and 2) that is compared to the input (i). The first match has its statement or block evaluated, and everything else is skipped. If there are no matches, the else statement or block is executed. The -> separates the branch comparison from the stuff that should be executed if that branch comparison is met.

In this case, the output is:

```
and now for something completely different
```

While that snippet uses Int values for comparison, that is not a requirement — anything that can be compared using equality (==) can be used:

```
val thingy = "foo"

when (thingy) {
  "foo" -> println("something")
  "bar" -> println("something else")
  else -> println("and now for something completely different")
}
```

IF, WHEN, AND WHILE

(from ["when Works with Any Type" in the Klassbook](#))

However, you cannot mix types, unless `==` happens to work to compare them. This, for example, fails with a compile error, pointing out that `Int` and `String` are incompatible types:

```
val thingy = "foo"

when (thingy) {
  "foo" -> println("something")
  2 -> println("something else")
  else -> println("and now for something completely different")
}
```

Using when Instead Of `else if`

The `when` syntax from the preceding section is fairly similar to `switch` and `case`. However, `when` has many more options.

One option is to skip the parameter to the `when`. In that case, each branch has its own Boolean expression, and the first that evaluates to `true` is used. This results in a more direct analogue to `if/else if/else`:

```
val i = 3

when {
  i > 10 -> println("something")
  i > 2 -> println("something else")
  else -> println("and now for something completely different")
}
```

(from ["when Without an Expression" in the Klassbook](#))

Consolidating Multiple Branches

If you have a few values that all should route to the same branch, you can use a comma-delimited list instead of a single value for the branch comparison against the value supplied to `when`:

```
val thingy = "foo"

when (thingy) {
  "foo", "goo" -> println("something")
}
```

IF, WHEN, AND WHILE

```
"bar", "baz", "frobozz" -> println("something else")
else -> println("and now for something completely different")
}
```

(from ["when with Commas" in the Klassbook](#))

Kotlin will compare each of the comma-delimited values and if any of them match, that branch is used.

For numbers, you can use `..` syntax and `in` to set up a “range” and test against it:

```
val i = 3

when (i) {
    in 3..10 -> println("something")
    in 1..2 -> println("something else")
    else -> println("and now for something completely different")
}
```

(from ["when with Ranges" in the Klassbook](#))

However, these simplifications only work when you supply a value to `when` (`when(thingy)`), as opposed to leaving `when` empty and using boolean expressions for the branches.

Expressions As Branch Conditions

The value for comparison does not need to be a constant — any expression will also work:

```
val thingy = "bar"
val otherThingy = "BAR"

when (thingy) {
    "foo", "goo" -> println("something")
    otherThingy.toLowerCase() -> println("something else")
    else -> println("and now for something completely different")
}
```

(from ["when with Expressions for Branches" in the Klassbook](#))

Here, the second branch compares `thingy` to `otherThingy.toLowerCase()` — in this case, they are equal, and so that branch is used.

While

Just as `if` is fairly common in programming languages, so is `while` or some other loop that is based on evaluating a condition.

So we have `while` in Java:

```
int i = 0;

while (i < 10) {
    i++;
}
```

...and JavaScript:

```
var i = 0;

while (i < 10) {
    i++;
}
```

...and Ruby:

```
i = 0

while i < 10 do
    i += 1
end
```

Kotlin offers much the same thing:

```
var i = 0

while (i < 10) {
    i++
    println(i)
}
```

(from ["while" in the Klassbook](#))

However, `while` is not as common in Kotlin as it is in other languages. More often we use operations on collections (e.g., `forEach()`) rather than `while`. However, it exists, should you find the need for it.

If/When As Expressions

In Kotlin, `if` and `while` seem like they are fairly similar to their equivalents in other languages. `when` is a bit different but it still functions a lot like `switch` in Java, etc.

Where things start to get interesting is when you realize that `if` and `when` not only offer branching, but that they are expressions, no different than `2 + 2`.

Ternary Operator Replacement

For example, in Java, we sometimes use the “ternary operator”:

```
int foo = (bar > 10 ? 10 : bar)
```

Here, we are setting `foo` equal to `bar`, unless `bar` is over 10, in which case we cap `foo` at 10. The parenthetical expression returns what appears between the `?` and the `:` if the boolean expression on the left is true, otherwise it returns what appears after the `:`.

Kotlin lacks ternary support, but more than makes up for it by allowing `if` to serve in the same role:

```
val bar = 5
val foo = if (bar > 10) 10 else bar

println(foo)
```

(from ["if as an Expression" in the Klassbook](#))

This is the same business rule as in Java, just implemented using an `if`.

Our `if` can have full braces-wrapped code blocks, if we need them:

```
val bar = 5
val foo = if (bar > 10) {
    10
}
else {
    bar
}

println(foo)
```

(from ["if as an Expression, Using Blocks" in the Klassbook](#))

IF, WHEN, AND WHILE

The value of the last statement of the block is what that block evaluates to. In this case, each of those blocks is a single statement, but if we had several statements in a block, while they are all executed, the value from the last statement is used for the result (assuming that block is used, based on the conditional expression).

when Expressions

when can also be used as an expression:

```
val i = 3

val message = when (i) {
  in 3..10 -> "something"
  in 1..2 -> "something else"
  else -> "and now for something completely different"
}

println(message)
```

(from "[when as an Expression](#)" in the *Klassbook*)

Here, the when evaluates to a string, which is assigned to message and printed.

Technically, the branches of the if or when do not have to evaluate to the same type. So, this runs:

```
val i = 3

val message = when (i) {
  in 3..10 -> "something"
  in 1..2 -> 5
  else -> "and now for something completely different"
}

println(message)
```

However, you will get some compiler warnings:

```
warning: conditional branch result of type String is implicitly cast to Any
  in 3..10 -> "something"
      ^
warning: conditional branch result of type Int is implicitly cast to Any
  in 1..2 -> 5
```

IF, WHEN, AND WHILE

```
^
warning: conditional branch result of type String is implicitly cast to Any
else -> "and now for something completely different"
```

Here, the compiler is telling you that since the branches evaluated to different types, the implied type of `message` is `Any`. `Any` in Kotlin is roughly analogous to `Object` in Java — it is the root type of the class hierarchy. Every class in Kotlin eventually extends from `Any`. `Any` is the only common ancestor class of `Int` and `String`, which is why the compiler chose that as the type for `message`. However, the warning is there to hint to you that perhaps what you wrote is not what you really have in mind.

Else Required

Normally, `if` and `when` do not have to be “exhaustive”. Here, “exhaustive” does not mean that it makes you tired (that would be “exhausting”). Rather, “exhaustive” means that all possibilities are covered by some branch. So, this is exhaustive:

```
if (i>10) {
    // do something
}
else {
    // do something else
}
```

Here, no matter what the value of `i` is, one of the two branches will be taken.

This, however, is not exhaustive:

```
if (i>10) {
    // do something
}
```

Nor is this:

```
val i = 3

when {
    i > 10 -> println("something")
    i > 2 -> println("something else")
}
```

In these cases, there are values for `i` for which *none* of the branches is valid. In that case, the `if` or `when` simply does not do anything.

IF, WHEN, AND WHILE

For standalone `if` and `when` statements, this is fine. However, if you are going to use `if` and `when` as expressions, they must be exhaustive. After all, we are trying to evaluate the value of the `if` or `when`, and we do not have a value if there is some input for which no branch qualifies.

If you try a non-exhaustive `if` or `when` for an expression, such as:

```
val i = 3

val message = when (i) {
  in 3..10 -> "something"
  in 1..2 -> 5
}

println(message)
```

...you will get a compiler error:

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

Bustin' Out

Sometimes, while you are in a loop, you will want to change the normal flow of the loop. For example, you are in a `for` loop, and after the 5th item, you wish to exit the loop, perhaps in response to some signal (e.g., a cancellation flag was set).

You can always use `return` to exit the entire function that you are in. Beyond that, though, there are two loop control statements that you can use: `break` and `continue`. These work much like their counterparts in Java and other programming languages.

`break`

`break` says “abandon the loop”:

```
var i = 0;

while (i < 10) {
  i++

  if (i == 5) break
```

IF, WHEN, AND WHILE

```
println(i)
}
```

(from "[break Statements](#)" in the Klassbook)

This results in:

```
1
2
3
4
```

Once `i` is equal to 5, we break out of the loop. Since we are doing that before the `println()` call, we do not print 5 to the output.

`continue`

`continue` says “skip the rest of this pass through the loop, and move along to the next pass through the loop”:

```
var i = 0;

while (i < 10) {
  i++

  if (i==5) continue

  println(i)
}
```

(from "[continue Statements](#)" in the Klassbook)

This gives us:

```
1
2
3
4
6
7
8
9
10
```

We are going through the loop all 10 times, unlike in the break case. However, we are

IF, WHEN, AND WHILE

skipping part of the loop code via `continue` — specifically, we are skipping over the `println()` call. As a result, 5 does not appear in the output, though all the other values do.

Object Orientation

Basic Classes

Because Kotlin lets you just write and invoke functions, you can accomplish a fair bit without actually creating your own classes. But, eventually, you will want to create classes, to model data and related behavior.

Basic Classes

Classes in Kotlin are very similar to their Java and Ruby counterparts: they primarily contain properties and functions that operate on those properties.

A class does not have to contain any of that — this is a valid Kotlin class definition:

```
class Foo {  
}
```

So is this:

```
class Foo
```

Having a class with no body seems odd. However, in Kotlin, you will occasionally see that sort of thing, particularly with [data classes](#) and [sealed classes](#). However, most ordinary classes will have properties and functions, as we will see below.

Creating Instances of Classes

Java and Ruby both use `new` to create instances of classes, whether as a keyword in Java:

BASIC CLASSES

```
Foo foo = new Foo();
```

...or as a method in Ruby:

```
foo = Foo.new
```

JavaScript has a few flavors of creating objects, including one using the `new` keyword, much as you would in Java.

In Kotlin, though, you treat the class name as a function name, and just call it:

```
class Foo

fun main() {
    val foo = Foo()

    println(foo)
}
```

(from "[Classes and Instances](#)" in the *Klassbook*)

In effect, you are calling the class “constructor” this way — we will explore constructors more [later in this chapter](#).

The `println()` call does not print much:

```
[object Object]
```

That is partly due to the limited built-in implementation of the `toString()` function that we inherit, and partly due to limitations in the Kotlin scripting environment used by *Klassbook*.

Packages

Kotlin makes use of packages the same way that Java does: as a namespace for classes. And, as with Java, by default a class is not in a package. Most Kotlin files in an actual project — as opposed to little REPL experiments — will be in a package.

The syntax for setting up a Kotlin package is nearly identical to that of Java: put a package line at the top of the file:

```
package bar.goo

class Foo

val foo = Foo()
```

What differs, compared to Java, is just the lack of the semicolon at the end of the package statement that the equivalent Java package statement would require.

Note that while having packages is *very* common in Kotlin projects, Klassbook does not use them, and neither do most other REPLs.

Common Contents

While Kotlin classes can hold lots of different sorts of programming constructs, the two most common are functions and properties. Just as we can define them outside of a class, we can define them inside of a class, for use by instances of that class.

Functions

Declaring an ordinary function is simply a matter of having the fun be inside the body of a class:

```
class Foo {
    fun something() {
        println("Hello, world!")
    }
}
```

Calling an object's function is accomplished by using dot (.) notation:

```
class Foo {
    fun something() {
        println("Hello, world!")
    }
}

fun main() {
    val foo = Foo()

    foo.something()
}
```

(from "[Adding Functions to Classes](#)" in the Klassbook)

You have access to the full range of capabilities that we describe in [the chapter on functions](#): parameters, varargs, expression bodies, local variables, etc.

Properties

Just as classes can have functions, they can have properties:

```
class Foo {
    var count = 0

    fun something() {
        count += 1
        println("something() was called $count times")
    }
}

fun main() {
    val foo = Foo()

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Adding Properties to Classes](#)" in the *Klassbook*)

Here, we have a count property, initially set to 0, that we increment on each call to something(). That count is then used in the printed output, courtesy of a bit of string interpolation.

Functions in the class can refer to properties just using the property name. Functions outside of the class, such as main(), need to use dot notation, just like we do for calling functions on an object.

So, we get four lines of output: three from the something() function that we are calling, and one from main() itself:

```
something() was called 1 times
something() was called 2 times
something() was called 3 times
the final count was 3
```

However, properties in Kotlin can get surprisingly complicated. We will explore

them in greater detail [in an upcoming chapter](#).

this

As with Java, Kotlin uses `this` as a pseudo-property to refer to the instance of the object in whose function you happen to be running.

You can use that as a prefix for property references or function calls, such as `this.count`:

```
class Foo {
    var count = 0

    fun something() {
        this.count += 1
        println("something() was called $count times")
    }
}

fun main() {
    val foo = Foo()

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[The this Pseudo-Property](#)" in the *Klassbook*)

Or you could use it on its own to refer to the current instance, such as for supplying that object as a parameter to a function:

```
class Foo {
    fun printMe() {
        println(this)
    }
}

fun main() {
    val foo = Foo()

    foo.printMe()
}
```

(from ["Passing this as a Parameter" in the Klassbook](#))

Occasionally, you will see odd variants of this, such as `this@Foo`. We will see what that syntax means [later in the book](#).

Constructors

As with Java and Ruby, a Kotlin class does not necessarily need an explicitly-declared constructor. By default, you get a zero-parameter constructor, as `Foo` did in the preceding examples. Many of your Kotlin classes will not need any custom constructors... but a few will need one, and on occasion they might need more than one.

What You Normally See

In Java and Ruby, declaring constructors is a matter of having a specific method (or thing that looks like a method) declared on the class. In Java, that pseudo-method has the name of the class:

```
class Foo {  
    final private int count;  
  
    Foo(int count) {  
        this.count = count;  
    }  
}
```

while in Ruby, we use the `initialize` method:

```
class Foo  
    def initialize(count)  
        @count = count  
    end  
end
```

Similarly, JavaScript uses a function named after the class, at least in some approaches of implementing objects in JavaScript.

The way you will normally see Kotlin classes have a constructor is by making the actual class name itself take the constructor parameters:

BASIC CLASSES

```
class Foo(var count: Int) {
    fun something() {
        count += 1
        println("something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Constructors](#)" in the Klassbook)

Here, we have one constructor parameter, `count`. The `var` keyword in front of it means that we can use it like a `var` property, in this case changing its value via the increment (`+=`) operator.

The Formal Approach

The constructor syntax shown above is really a shorthand for using the constructor keyword:

```
class Foo constructor(var count: Int) {
    fun something() {
        count += 1
        println("something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Formal Constructor Declaration](#)" in the Klassbook)

For most classes, with zero or one constructors, you will not see the constructor keyword used. It becomes more important when you have [multiple constructors](#), including [private constructors](#).

Init Blocks

Compared with Java and Ruby constructors, though, our Kotlin constructors are missing something: a body. All we are doing is declaring a list of parameters that can be supplied to the instance of the class.

Kotlin instead uses an init block.

Syntax-wise, an init block resembles a Java static block: a set of braces-wrapped statements preceded by a keyword. In Java, the static block is executed once for the class and is used to initialize complex static fields. In Kotlin, an init block is executed once for each *instance* and serves as a baseline constructor body:

```
class Foo(val sillyCount: String) {
    var count = 0

    init {
        count = sillyCount.toInt()
    }

    fun something() {
        count += 1
        println("something() was not called $count times")
    }
}

fun main() {
    val foo = Foo("7")

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Init Block](#)" in the [Klassbook](#))

Here, inexplicably, we are passing in a `String` representation of an initial value to use for the count, rather than passing in the `var count` as we did before. The `init`

block is used to initialize count from sillyCount as part of setting up our instance of Foo. There is a simpler way to handle this, via property initialization, that we will explore more in [the chapter on properties](#).

init blocks have a couple of other distinctions, particularly when compared to Java/Ruby constructors:

- You can have more than one init block. They are executed in the order that they appear in the class.
- An init block can only reference properties that are declared before it in the class, not ones declared after it. As a result, an init block often appears after the list of properties, so the init block can reference any of the properties.

Parameters Sans var Or val

Sometimes, you will see constructor parameters that do not have the var or val keyword. They just have the parameter name and type. They are still constructor parameters, but they are not properties. You cannot refer to them from the functions of your class. You can refer to them from init blocks, though:

```
class Foo(sillyCount: String) {
    var count = 0

    init {
        count = sillyCount.toInt()
    }

    fun something() {
        count += 1
        println("something() was not called $count times")
    }
}

fun main() {
    val foo = Foo("7")

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Constructor Init-Only Parameters](#)" in the *Klassbook*)

This is the same as the preceding example, just without the `var`. It works, because the only place we refer to `sillyCount` is from an `init` block.

Having Multiple Constructors

Some Kotlin classes have no constructor, using only the default zero-parameter constructor. Some Kotlin classes have one constructor. And a handful of classes will have more than one constructor.

The primary constructor appears in the class declaration itself, as we have seen. Other constructors, if they exist, are called “secondary constructors”, and they somewhat resemble ordinary functions:

```
class Foo(var count: Int) {  
  
    constructor(sillyCount: String) : this(sillyCount.toInt()) {  
        // could have code here if needed  
    }  
  
    fun something() {  
        count += 1  
        println("something() was not called $count times")  
    }  
}  
  
fun main() {  
    val foo = Foo("7")  
  
    foo.something()  
    foo.something()  
    foo.something()  
  
    println("the final count was ${foo.count}")  
}
```

(from "[Secondary Constructors](#)" in the *Klassbook*)

A primary constructor has no function body — any such code goes into an `init` block. Secondary constructors, though, can have bodies, the way that functions do, but only if needed.

Secondary constructors must arrange to call the primary constructor. This is handled by having a `this()` call after the constructor parameter list, separated from that list via a colon. The `this()` call needs to match the parameter list of the

primary constructor — in the example shown above, the secondary constructor needs to provide a `Int` to the primary constructor.

Default Parameter Values

Having multiple constructors is comparatively common in a language like Java, as through Java 8, there was no support for default parameter values. Hence, if you wanted to allow for both simple and complex object instantiation, you needed to declare all of the necessary constructors, perhaps chaining from one to another. For example, Android developers who have created custom `View` classes might be familiar with the four `View` constructors:

```
public View(Context context) {
    // a ton of code
}

public View(Context context, @Nullable AttributeSet attrs) {
    this(context, attrs, 0);
}

public View(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {
    this(context, attrs, defStyleAttr, 0);
}

public View(Context context, @Nullable AttributeSet attrs, int defStyleAttr,
            int defStyleRes) {
    this(context);

    // another ton of code
}
```

Here, each of the constructors simply adds on another possible parameter to be supplied, and the constructors chain to other constructors to eliminate duplication of initialization logic.

Kotlin, like Ruby, dispenses with this, by allowing for you to define default values for your constructor parameters... just as you can on [ordinary functions](#). The Kotlin equivalent of the `View` constructor set might look like:

```
class View(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0,
    defStyleRes: Int = 0
```

```
) {  
    // two tons of code  
}
```

Here, the latter three parameters all have default values defined, giving us the ability to call the constructor with one parameter, all four parameters, or combinations in between.

The above code snippet contains newlines in between the parameters. That is merely a matter of code formatting. Typical Kotlin will split long parameter lists up this way, with one parameter per line.

Also, `attrs` is shown as having a type of `AttributeSet?`. As was mentioned previously (briefly), the `?` suffix indicates that this is a “nullable type”, and we will explore that in greater detail in [an upcoming chapter](#).

Inheritance

As with many object-oriented languages, Kotlin supports class inheritance. A subclass inherits from a superclass, and what it “inherits” are all of the properties, functions, constructors, and other stuff that the superclass makes available.

What Happens By Default

The various versions of the `Foo` class shown above do not inherit explicitly from anything. The default base class for a Kotlin class is `Any`, and so `Foo` is said to derive from `Any`.

Extending a Class

In Java, we use the `extends` keyword to denote inheritance:

```
class Foo extends Bar {  
  
}
```

Ruby uses the `<` symbol:

```
class Foo < Bar  
  
end
```

BASIC CLASSES

Kotlin is a bit closer to Ruby syntax, but it uses a colon as the symbol:

```
class Foo(var count: Int) : Any() {
    fun something() {
        count += 1
        println("something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Inheritance](#)" in the *Klassbook*)

Here, we are explicitly declaring that the superclass of Foo is Any.

Chaining to the Superclass Constructors

The () at the end of Any() in the above example is actually a call to the superclass constructor. You will need to supply parameters to that call, if the superclass requires them:

```
open class Base(val tag: String)

class Foo(var count: Int) : Base("Foo Example") {
    fun something() {
        count += 1
        println("$tag: something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()
}
```

BASIC CLASSES

```
println("the final count was ${foo.count}")
}
```

(from "[Inheritance and Constructors](#)" in the *Klassbook*)

Here, Base has a constructor requiring a tag parameter. Foo needs to supply that parameter when it calls the Base() constructor. But then Foo can reference the tag property that it inherits, including it as part of the printed output:

```
Foo Example: something() was called 1 times
Foo Example: something() was called 2 times
Foo Example: something() was called 3 times
the final count was 3
```

Open Classes

You will notice that the declaration of Base has something beyond a constructor parameter: it is prefixed with the open keyword.

In Java, by default, any class can be extended by a subclass. To prevent a class from being extended, you have to mark it as final.

Kotlin goes the opposite route. Kotlin classes *cannot* be extended by default. You have to mark classes that can be extended using open. Failure to do that results in an error. So, if we remove the open from the previous example:

```
class Base(val tag: String)

class Foo(var count: Int) : Base("Foo Example") {
    fun something() {
        count += 1
        println("$tag: something() was called $count times")
    }
}

val foo = Foo(0)

foo.something()
foo.something()
foo.something()
```

...we get a compile error:

BASIC CLASSES

```
error: this type is final, so it cannot be inherited from
class Foo(var count: Int) : Base("Foo Example") {
    ^
```

This has an important side effect: you cannot create subclasses of arbitrary classes that you do not control. Only if the implementer of that other class is willing to support subclasses will the implementer use the open keyword, and only those classes can you extend.

This approach is uncommon in major programming languages. Basically, Kotlin recognizes that inheritance is a potentially fragile relationship between two classes. Ideally, the classes adhere to [the Liskov substitution principle](#). This basically states that instances of a subclass should be usable *anywhere* that instances of the superclass can be used. A programming language cannot guarantee that, and developers sometimes get sloppy when creating subclasses. So, Kotlin forces developers to make a conscious decision to allow subclasses, in hopes that it helps those developers take appropriate defensive programming steps to ensure that subclasses behave as expected.

Overriding Functions

As you might expect, a subclass can call functions that it inherits from its superclass:

```
open class Base(val tag: String) {
    fun gimmeTheTag() = tag
}

class Foo(var count: Int) : Base("Foo Example") {
    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Inheritance and Functions](#)" in the *Klassbook*)

BASIC CLASSES

And, as you might expect, a subclass can override functions of its superclasses... at least, if those too are marked as open. By default, a Kotlin function is not open and cannot be overridden. Not only must the superclass function be marked as open, but the *overriding* function must be marked with `override`:

```
open class Base(val tag: String) {
    open fun gimmeTheTag() = tag
}

class Foo(var count: Int) : Base("Foo Example") {
    override fun gimmeTheTag() = "Not the Original Tag"

    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Inheritance and Overriding Functions](#)" in the *Klassbook*)

Here, `gimmeTheTag()` is an open function in `Base`, so `Foo` can override it with its own implementation. The string interpolation will use the overridden function:

```
Not the Original Tag: something() was called 1 times
Not the Original Tag: something() was called 2 times
Not the Original Tag: something() was called 3 times
the final count was 3
```

By default, `open` is transitive. Once a function is marked as open, it is open for all subclasses down the hierarchy, regardless of whether intervening classes override the function or not:

```
open class Base(val tag: String) {
    open fun gimmeTheTag() = tag
}
```

BASIC CLASSES

```
open class Intermezzo(val thing: String) : Base(thing)

class Foo(var count: Int) : Intermezzo("Foo Example") {
    override fun gimmeTheTag() = "Not the Original Tag"

    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[override is Transitive](#)" in the *Klassbook*)

Here, even though Foo now extends Intermezzo, it can still override gimmeTheTag(). It could do so even if Intermezzo itself overrides gimmeTheTag()... unless Intermezzo marks its version of that function as final:

```
open class Base(val tag: String) {
    open fun gimmeTheTag() = tag
}

open class Intermezzo(val thing: String): Base(thing) {
    final override fun gimmeTheTag() = "-$thing-"
}

class Foo(var count: Int) : Intermezzo("Foo Example") {
    override fun gimmeTheTag() = "Not the Original Tag"

    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

val foo = Foo(0)
```

```
foo.something()
foo.something()
foo.something()
```

This results in an error:

```
error: 'gimmeTheTag' in 'Intermezzo' is final and cannot be overridden
  override fun gimmeTheTag() = "Not the Original Tag"
  ^
```

Once again, the objective is to force everybody to think about the ramifications of allowing functions to be overridden and the ramifications of overriding those functions.

Overriding Properties

It is also possible for a class to declare a *property* as open, allowing subclasses to override it.

Mostly this is needed for properties declared as constructor parameters. For example, let's go back to this script:

```
open class Base(val tag: String) {
    open fun gimmeTheTag() = tag
}

open class Intermezzo(val thing: String) : Base(thing)

class Foo(var count: Int) : Intermezzo("Foo Example") {
    override fun gimmeTheTag() = "Not the Original Tag"

    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()
}
```

BASIC CLASSES

```
println("the final count was ${foo.count}")
}
```

(from "[override is Transitive](#)" in the *Klassbook*)

Note that the constructor parameter for Base is called tag, while the constructor parameter for Intermezzo is called thing. They happen to be the same, um, thing, since Intermezzo passes its thing property to the call to the Base constructor.

Suppose, though, that we wanted to name these the same, such as having them both be tag:

```
open class Base(val tag: String) {
    open fun gimmeTheTag() = tag
}

open class Intermezzo(val tag: String): Base(tag)

class Foo(var count: Int) : Intermezzo("Foo Example") {
    override fun gimmeTheTag() = "Not the Original Tag"

    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

val foo = Foo(0)

foo.something()
foo.something()
foo.something()
```

This results in a compile error:

```
error: 'tag' hides member of supertype 'Base' and needs 'override' modifier
open class Intermezzo(val tag: String): Base(tag)
                        ^
```

This is because both are declared as properties, and by default you cannot re-declare a property by reusing the name.

One way to address this is to realize that we do not need the constructor parameter in Intermezzo to be a property. It could be a plain constructor parameter:

BASIC CLASSES

```
open class Base(val tag: String) {
    open fun gimmeTheTag() = tag
}

open class Intermezzo(tag: String): Base(tag)

class Foo(var count: Int) : Intermezzo("Foo Example") {
    override fun gimmeTheTag() = "Not the Original Tag"

    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

val foo = Foo(0)

foo.something()
foo.something()
foo.something()
```

This works, even though we reuse the tag name in Intermezzo, because a plain tag constructor parameter does not conflict with an inherited val property, even one that itself is declared as a constructor parameter.

What the compile error suggests, though, is that we can make tag in Base be open, then override it in Intermezzo:

```
open class Base(open val tag: String) {
    open fun gimmeTheTag() = tag
}

open class Intermezzo(override val tag: String) : Base(tag)

class Foo(var count: Int) : Intermezzo("Foo Example") {
    override fun gimmeTheTag() = "Not the Original Tag"

    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
}
```

BASIC CLASSES

```
foo.something()
foo.something()

println("the final count was ${foo.count}")
}
```

(from "[Overriding Properties](#)" in the Klassbook)

In other words, you can override properties and change their behavior — we will examine this more [much later in the book](#).

Chaining to Superclass Functions

Like Java, Kotlin uses a `super .` prefix to allow code in a subclass to specifically call implementations in a superclass. Typically you do this in a function that you overrode, to execute the superclass implementation as part of the overridden function's implementation.

```
open class Base(val tag: String) {
    open fun gimmeTheTag() = tag
}

class Foo(var count: Int) : Base("Foo Example") {
    override fun gimmeTheTag() = super.gimmeTheTag() + "-Extended"

    fun something() {
        count += 1
        println("${gimmeTheTag()}: something() was called $count times")
    }
}

fun main() {
    val foo = Foo(0)

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Chaining to Superclass Functions](#)" in the Klassbook)

Here, the `Foo` implementation of `gimmeTheTag()` calls the superclass implementation and appends `-Extended` to it. As a result, we get the combined results of both the `Base` and the `Foo` implementations of `gimmeTheTag()`:

BASIC CLASSES

```
Foo Example-Extended: something() was called 1 times
Foo Example-Extended: something() was called 2 times
Foo Example-Extended: something() was called 3 times
the final count was 3
```

Inheritance Tests

Sometimes, we need to know whether a reference to some supertype is an instance of some subtype.

In Java, usually we do this via `instanceof`. So if we have a class hierarchy like this:

```
class Animal {
    // stuff
}

class Frog extends Animal {
    // froggy stuff
}

class Axolotl extends Animal {
    // axolotly stuff
}
```

...we can use `instanceof` to see if some `Animal` is really a `Frog`:

```
void something(Animal critter) {
    if (critter instanceof Frog) {
        // do froggy things
    }
}
```

In Ruby, `kind_of?` or `is_a?` are the typical ways of accomplishing the same thing:

```
def something(critter)
    if critter.is_a? Frog
        # do froggy things
    end
end
```

Kotlin has syntax similar to Java, with a shorter keyword: `is`. So, we can do this:

```
open class Animal
class Frog : Animal()
```

BASIC CLASSES

```
class Axolotl : Animal()

fun main() {
    val critter: Animal = Frog()

    if (critter is Frog) println("Ribbit!") else println("Ummm... whatever noise an axolotl makes!")
}
```

(from "[Checking Inheritance Via is](#)" in the *Klassbook*)

Here, `critter` is an `Animal`, but we use `is` to determine whether it is really pointing to an instance of `Frog` or not.

Inheritance and `when`

In Kotlin, we can use `is` not only as part of an expression, but we can use it as a test for a `when` clause. We can rewrite the previous example using a `when`:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val critter: Animal = Frog()

    when (critter) {
        is Frog -> println("Ribbit!")
        else -> println("Ummm... whatever noise an axolotl makes!")
    }
}
```

(from "[is and when](#)" in the *Klassbook*)

This is helpful if you have a series of subtypes to check and handle separately, though you may be better served by overriding functions and putting the logic in the classes themselves (e.g., have a `makeSound()` function that all `Animal` classes implement, so you can just call `makeSound()` on any `Animal` and get the desired result).

Casting

Java programmers have a long history with casting objects. This turns a reference to a supertype into a subtype. In Java, we use the subtype name in parentheses to perform the cast. So, if we have the `Animal`, `Frog`, and `Axolotl` types as shown above,

we might have a method that looks like this:

```
void something(Animal critter) {
    if (critter instanceof Frog) {
        Frog frog = (Frog)critter;

        // do something with the frog, preferably not involving boiling water
    }
    else {
        Axolotl axolotl = (Axolotl)critter;

        // do... oh, I don't know, I'm sure you can think of something
    }
}
```

In Kotlin, casting gets a bit more interesting, because the compiler tries to help reduce the number of casts that you need. In fact, well-written Kotlin code rarely involves casting.

Manual

That being said, you can certainly cast types. In Kotlin, that is performed by using the `as` keyword:

```
val critter: Animal = Frog()
val kermit = critter as Frog
```

Here, `kermit` will be of type `Frog`, as we manually cast the `Animal` to `Frog` via `as`.

Smart Casts

Let's go back to the Java snippet for a moment:

```
void something(Animal critter) {
    if (critter instanceof Frog) {
        Frog frog = (Frog)critter;

        // do something with the frog, preferably not involving boiling water
    }
    else {
        Axolotl axolotl = (Axolotl)critter;
    }
}
```

BASIC CLASSES

```
    // do... oh, I don't know, I'm sure you can think of something
  }
}
```

In the `if` block, we cast `critter` to be a `Frog`. We know this is safe, because we checked the type validity with `instanceof` in the `if` test.

So, if we know that `critter` is really a `Frog`... why doesn't Java?

Presumably, the Kotlin developers asked themselves that sort of question, though possibly not involving frogs. Inside of `if` and `when` blocks, if we know that a variable is really of some subtype, the compiler lets us skip the casts and allows us to refer to the variable as the subtype, even though it was declared as a supertype.

For example, let's give our creatures some functions, then call those functions:

```
open class Animal

class Frog : Animal() {
    fun hop() = println("Hop!")
}

class Axolotl : Animal() {
    fun swim() = println("Swish!")
}

fun main() {
    val critter: Animal = Frog()

    when (critter) {
        is Frog -> critter.hop()
        is Axolotl -> critter.swim()
    }
}
```

(from ["Smart Casts" in the Klassbook](#))

In a Java-style world, this would not compile. `critter` is an `Animal`, and `Animal` has neither `hop()` nor `swim()`. However, Kotlin realizes that the `is Frog` code will only be executed if `critter` is a `Frog`, so it allows us to call `hop()` on `critter`, as if `critter` were declared as a `Frog` instead of as an `Animal`. Similarly, we can call `swim()` on `critter` if it is `Axolotl`.

However, this only works when the compiler is *certain* of the type. As a result, this subtly-different edition of that code fails with a compile error:

BASIC CLASSES

```
open class Animal

class Frog : Animal() {
    fun hop() = println("Hop!")
}

class Axolotl : Animal() {
    fun swim() = println("Swish!")
}

val critter: Animal = Frog()

when (critter) {
    is Frog -> critter.hop()
    else -> critter.swim()
}
```

Here, we replace `is Axolotl` with `else`. The Kotlin compiler knows that `critter` is not a `Frog`, as otherwise we would not be executing the `else`. However, `critter` might not be an `Axolotl`, as it could be an instance of `Animal`. As a result, Kotlin cannot assume that it is safe to call `swim()` on `critter`, as while an `Axolotl` can `swim()`, an `Animal` cannot. So, we get a compile error:

```
error: unresolved reference: swim
    else -> critter.swim()
```

To get it to compile, we need to manually cast the `critter` to be an `Axolotl`:

```
open class Animal

class Frog : Animal() {
    fun hop() = println("Hop!")
}

class Axolotl : Animal() {
    fun swim() = println("Swish!")
}

fun main() {
    val critter: Animal = Frog()

    when (critter) {
        is Frog -> critter.hop()
        else -> (critter as Axolotl).swim()
    }
}
```

BASIC CLASSES

(from "[Manual Casts](#)" in the *Klassbook*)

But, by doing this, we might crash, if `critter` is neither a `Frog` nor an `Axolotl`. So, while this compiles:

```
open class Animal

class Frog : Animal() {
    fun hop() = println("Hop!")
}

class Axolotl : Animal() {
    fun swim() = println("Swish!")
}

val critter: Animal = Animal()

when (critter) {
    is Frog -> critter.hop()
    else -> (critter as Axolotl).swim()
}
```

...it crashes at runtime, such as this error message from Kotlin/JVM:

```
java.lang.ClassCastException: Test$Animal cannot be cast to Test$Axolotl
at Test.<init>(Unknown Source)
```

Kotlin's "smart casts" — as this automatic casting is called — not only saves you typing, but it also helps you avoid improper manual casts. In general, if you find yourself using `as` in Kotlin, you should be asking yourself: if the Kotlin compiler does not know that this object is of this subtype... how do *I* know that this object will always be of this subtype?

Comments and Documentation

Now that we are starting to assemble Kotlin classes, it is time to address the two things that developers seem to dislike most: testing and documentation.

How you write tests for your Kotlin code will depend a lot on the environment in which you are running that code. So, for example, Android app developers using Kotlin will still write JUnit tests, just using Kotlin instead of Java. As a result, there is little that a book on the Kotlin *language* can tell you about testing Kotlin, which is why you will see little on this topic here.

However, when it comes to documenting your Kotlin code, Kotlin supports code comments, much like most programming languages. And, just as Java has JavaDocs and Ruby has RDoc and so forth, Kotlin has KDoc for creating comments that can be turned into documentation.

Basic Comment Syntax

Kotlin adopted Java comment syntax, which happens to line up with JavaScript comment syntax.

// indicates that the rest of the line is a comment:

```
val foo = 0    // this is a comment
// so is this
```

/* and */ are used to delimit multiple lines as being a comment:

COMMENTS AND DOCUMENTATION

```
val foo = 0    // this line has executable code

/*
val bar = 0    // this line does not, as it is wrapped in a comment
*/
```

The biggest — and perhaps only — difference between Kotlin comments and what you might do in Java is that multi-line comments can be nested in Kotlin. So, for example, this is valid in Kotlin:

```
/*
  It's
  /*
  comments
  /*
  all
  /*
  the
  /*
  way
  /*
  down!
  */
  */
  */
  */
  */
  */
  */
```

Introducing KDoc

If all that you want to do is add explanations to your code, to be read as part of that code, the comment syntax shown above is all that you need.

If you want to be able to generate standalone documentation for your code, the way that JavaDocs, RDoc, and similar tools do... Kotlin's equivalent is KDoc.

Basic KDoc Comments

Once again, Kotlin uses Java's approach and extends it. So, just as JavaDoc comments start with `/**` and end with `*/`, so too do KDoc comments:

COMMENTS AND DOCUMENTATION

```
/**
 * This is the summary of your Kotlin class. It should explain, in a single
 * paragraph, the role of this class in your project.
 *
 * Additional paragraphs after the first one should explain your class in
 * greater detail. Of course, this class is extremely simple. It does not
 * need much in the way of explanation. Yet, for some reason, the author
 * insists on keeping on typing text into this paragraph, as if his hand is
 * guided by some unseen force. Or, perhaps, he is just trying to pad the
 * paragraph out a bit.
 *
 * Oh, by the way, the line of asterisks that you see on the left is optional,
 * as you will see in the function comment below.
 */
class Foo {

    /**
     * This should explain what this function does.
     *
     * Um, that's it!
     */
    fun something() {
        println("Hello, world!")
    }
}
```

Formatting

Where KDoc breaks from JavaDoc is in terms of text formatting. KDoc uses [Markdown](#), arguably the world's most popular “wikitext” markup language. Everything from Stack Overflow questions to GitHub README files are written in Markdown. This book is written in Markdown. And KDoc lets you use Markdown formatting in your comments.

So, you can use things like:

- ****Boldface**** (renders as: **Boldface**)
- **Italics** (renders as: *Italics*)
- Backticks wrapped around text to format it in monospace
- And so on

Cross-Referencing

What Markdown lacks is a way to link to things inside the same document. Its

COMMENTS AND DOCUMENTATION

hyperlink syntax works great for links to other pages, but not to material in the current page.

KDoc extends Markdown linking syntax, allowing you to reference properties, functions, and similar named things by using simple bracket syntax:

```
/**
 * This is the summary of your Kotlin class. It should explain, in a single
 * paragraph, the role of this class in your project.
 *
 * This class has a single function: [something].
 *
 * You can also link to other classes, such as [kotlin.String], using the
 * fully-qualified class name.
 *
 * And, of course, you can [link to external content](https://commonsware.com).
 */
class Foo {

    /**
     * This should explain what this function does.
     *
     * This function is inside of [Foo], as you can tell by looking up a few lines
     * from where you are reading right now.
     */
    fun something() {
        println("Hello, world!")
    }
}
```

Block Tags

As with JavaDoc, KDoc supports “block tags” for documenting sub-elements of class, function, etc. For example, you can use `@param` to document a function parameter, `@return` to document a function return value, and so forth:

```
/**
 * This is the summary of your Kotlin class. It should explain, in a single
 * paragraph, the role of this class in your project.
 *
 * This class has a single function: [something].
 *
 * You can also link to other classes, such as [kotlin.String], using the
 * fully-qualified class name.
 *
 * @author Mark Murphy
 */
```

COMMENTS AND DOCUMENTATION

```
*/  
class Foo {  
  
    /**  
     * This should explain what this function does.  
  
     * This function is inside of [Foo], as you can tell by looking up a few lines  
     * from where you are reading right now.  
  
     * @param msg the message to be printed  
     * @return `true`, because, hey, why be negative?  
     */  
    fun somethingMoreElaborate(msg: String = "Hello, world!"): Boolean {  
        println(msg)  
  
        return true  
    }  
}
```

A complete list of block tags is available in [the Kotlin documentation](#).

Generating the Documentation

Just as Java has a javadoc program to generate JavaDocs, and just as Ruby has rdoc to generate documentation from RDoc comments, Kotlin has a similar tool: [Dokka](#).

Most likely, you will generate your documentation using the build system for your project, such as Gradle for an Android app project. There is a way to [generate the documentation from the command line](#), though Dokka does not seem to be optimized for that use case.

Properties

In [the chapter on classes](#), we saw that we can have properties in our Kotlin classes, reminiscent of having fields in Java:

```
class Foo {
    var count = 0

    fun something() {
        count += 1
        println("something() was called $count times")
    }
}

fun main() {
    val foo = Foo()

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Adding Properties to Classes](#)" in the *Klassbook*)

In a nutshell, you can have var- and val-labeled properties in your classes, just like you can outside of classes. Properties in classes are accessible by instances of those classes, just as the something() function in Foo has access to the count property.

So far, this is unremarkable, and you might wonder why there is a whole chapter dedicated to properties. That is because properties in Kotlin have a number of features that go beyond their equivalents in other popular programming languages. These features are commonly used in Kotlin development, and so it is important to

understand what they are and how you use them.

Initialization

Properties need to be initialized at some point. Kotlin gives you some common options, and some less-common options, for doing this.

When Declared

Much of the time, you will initialize properties when you declare them. Primarily, such initialization will refer to literals, constructor parameters, properties declared outside of classes, and perhaps [constants](#). They can also refer to other properties that themselves are initialized.

Here, we initialize count based on a sillyCount constructor parameter:

```
class Foo(sillyCount: String) {
    var count = sillyCount.toInt()

    fun something() {
        count += 1
        println("something() was not called $count times")
    }
}

fun main() {
    val foo = Foo("7")

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Property Initialization in Declaration](#)" in the *Klassbook*)

In an Init Block

You can also initialize properties in an init block. In [the chapter on classes](#), we saw this example:

PROPERTIES

```
class Foo(val sillyCount: String) {
    var count = 0

    init {
        count = sillyCount.toInt()
    }

    fun something() {
        count += 1
        println("something() was not called $count times")
    }
}

fun main() {
    val foo = Foo("7")

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Init Block](#)" in the [Klassbook](#))

Here, we initialize count to 0, then replace it immediately with a value derived from sillyCount. As it turns out, the initialization is not required. This is just as valid and gives the same results:

```
class Foo(sillyCount: String) {
    var count: Int

    init {
        count = sillyCount.toInt()
    }

    fun something() {
        count += 1
        println("something() was called $count times")
    }
}

fun main() {
    val foo = Foo("7")

    foo.something()
}
```

PROPERTIES

```
foo.something()
foo.something()

println("the final count was ${foo.count}")
}
```

(from "[Property Initialization in Init Block](#)" in the *Klassbook*)

Sometime Later

It may be that you do not know how to initialize the property now, but you will be able to initialize it later. For example, in Android app development, there will be properties that you cannot initialize until some particular point in an object's lifecycle, such as in `onCreate()` of an Activity or Fragment.

The most common way of handling this is via `lateinit`. `lateinit` is a promise that you will not attempt to reference a property until you initialize it "late" (i.e., sometime after the object is initialized):

```
class Foo {
    var count = 0
    lateinit var label: String

    fun defineTag(tag: String) {
        label = tag
    }

    fun something() {
        count += 1
        println("$label: something() was called $count times")
    }
}

fun main() {
    val foo = Foo()

    foo.defineTag("Foo")
    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[lateinit Properties](#)" in the *Klassbook*)

PROPERTIES

Here, we have both a `count` and a `label`, where we use both in the printed output. For some reason, we do not have the value for `label` at the time we create an instance of `Foo`. So, we define `label` as `lateinit`, so we can skip the initialization.

The implied requirement is that `defineTag()` must be called before `something()`. We need to initialize `label` before we can refer to it safely. Failing to do this — such as by commenting out the `foo.defineTag("Foo")` line in the sample — results in a crash:

```
kotlin.UninitializedPropertyAccessException: lateinit property label has not been initialized
```

Where possible, you should try to avoid `lateinit`, as it is easy to make mistakes and try using the `lateinit` property before it is initialized.

Also note that `lateinit` cannot be used for types based on primitives, so you cannot have a `lateinit Int`, `Boolean`, etc.

Eh, Whenever

Yet another possibility is to use `by lazy`:

```
class Foo(val rawLabel: String) {
    var count = 0
    val label: String by lazy { println("initializing via lazy!"); rawLabel.toUpperCase() }

    fun something() {
        count += 1
        println("$label: something() was called $count times")
    }
}

fun main() {
    val foo = Foo("foo")

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Lazy Properties](#)" in the *Klassbook*)

Here, `label` is initialized lazily. Whenever we first attempt to use `label`, the lambda expression will be executed, and that value is what we see as the value of `label`. In this case, the lambda expression takes a constructor parameter and converts it into all caps, so our output is:

PROPERTIES

```
initializing via lazy!  
F00: something() was called 1 times  
F00: something() was called 2 times  
F00: something() was called 3 times  
the final count was 3
```

In truth, we would not need to use `by lazy` here. Since the constructor parameter is available at initialization time, and it is not changing, we could just initialize `label` normally:

```
class Foo(val rawLabel: String) {  
    var count = 0  
    val label = rawLabel.toUpperCase()  
  
    fun something() {  
        count += 1  
        println("$label: something() was called $count times")  
    }  
}  
  
val foo = Foo("foo")  
  
foo.something()  
foo.something()  
foo.something()
```

As with `lateinit var`, `by lazy` is good for cases where you cannot initialize the `val` up front, for whatever reason (e.g., `onCreate()` has not yet been called on your Android Activity). And, as with `lateinit var`, you run the risk of crashing if the conditions are not ready for your lambda expression to be evaluated.

The difference amounts to “push” versus “pull”. With `lateinit var`, we try to initialize the property as soon as it is practical, pushing a value into it. With `by lazy`, we hope that we do not try referencing it before the lambda expression will work... but once it is referenced, we “pull” its initial value from other data, using the lambda expression to derive the value that we want to use.

`by lazy` is one example of a broader pattern of property delegates, which we will explore [much later in the book](#).

Wait a Minute! Was That... a Semicolon?

Our `label` property not only had a lazy property delegate, but the lazy lambda expression had a rarity in Kotlin: a semicolon.

As with Java, a semicolon in Kotlin serves as a statement terminator. However, in Kotlin, a newline *also* can serve as a statement terminator, which is why we rarely see semicolons used in that role. In this case, the lazy lambda expression contains two statements: one to log a message mentioning that the lambda expression was being evaluated, and one to convert the string to uppercase.

Fake Properties

Another way to initialize a property... is to have it only look to outsiders like it is a regular property.

We usually distinguish between accessing a property and calling a function by the existence of parentheses. `anObject.someProperty` references a property, while `anObject.thatFunction()` calls a function.

In reality, Kotlin is not nearly that simple. It is possible to call functions without using parentheses — `forEach`, for example, is a function. Beyond that, there are ways to define properties such that they really are implemented by functions.

We will explore this way of setting up “fake properties” more [much later in the book](#).

Constants

Kotlin has a `const` keyword, which you can use to declare certain `val` properties as constants. This is somewhat reminiscent of `final` in Java.

This may seem a bit odd. After all, by many definitions, `val` *already* is a constant. You cannot change its value, after all.

There are some reasons for having `const`, though, but we will hold off on those until [much later in the book](#). For now, if while reading Kotlin code you see the `const` keyword, just imagine that it means “no, *really*, I mean that this should be a constant”.

Getting Down in the Weeds

In Java, we have fields:

PROPERTIES

```
class Foo {  
    private int myField;  
}
```

We make the comparison to Kotlin properties, but that is only an approximation.

In Java, sometimes we define “getter” and “setter” methods, and sometimes those adhere to “JavaBean” naming conventions:

```
class Foo {  
    private int myField;  
  
    int getMyField() {  
        return myField;  
    }  
  
    void setMyField(int newValue) {  
        myField = newValue;  
    }  
}
```

A Kotlin property really represents the combination of these three elements:

- a field
- a getter function
- a setter function

It so happens that for simple properties — like those we have seen in this chapter and previously — those three elements have default implementations that are hidden behind the simple var-style declaration. When we read a property’s value, we really are calling a getter function that returns the value of the property’s “backing field”. When we write a property’s value, we really are calling a setter function that sets the value of the backing field.

Some advanced Kotlin syntax — such as the “fake properties” alluded to earlier — take advantage of this, overriding the default getter and/or setter to do something else.

For most Kotlin programming, though, this is “a distinction without a difference”. Property syntax resembles accessing Java fields, so we think of properties as being equivalent to fields. However, if you later advance to the “WTF?” chapters of this book, we will return to this concept of properties really being fields with getter/setter functions.

Visibility and Scope

Now that we are starting to define classes with properties and functions, we need to start thinking about “who can see what” — what particular aspects of our code are accessible by other aspects of our code. Principally, that amounts to two things:

1. Visibility: What is allowed to see these classes, functions, and properties?
2. Scope: Where do these classes, functions, and properties actually exist?

Visibility

Many programming languages have visibility options, to control what objects can reference functions and properties from other objects. In both Java and Ruby, for example, you can have public, private, and protected methods. Kotlin has those too, though Kotlin adheres a bit closer to Java’s definition of those terms than Ruby’s. Kotlin also has an `internal` visibility option that is a little strange.

Default = Public

In some languages, everything is public by default. Any object can reference public things. In Java, though, everything is “package-private” by default, meaning that it is public to things in the same Java package, but private to things in other packages.

Kotlin takes the public-by-default route. You will not see a `public` keyword used in typical Kotlin. Everything that is defined by default is visible to anything else. This is why code like this — seen in [the chapter on classes](#) — actually works:

```
class Foo {  
    fun something() {  
        println("Hello, world!")  
    }  
}
```

VISIBILITY AND SCOPE

```
}  
}  
  
fun main() {  
    val foo = Foo()  
  
    foo.something()  
}
```

(from "[Adding Functions to Classes](#)" in the *Klassbook*)

Both `Foo` and `something` are public, and so code that lives outside of `Foo` can reference `Foo` (for creating instances) and `something()` (for calling that function on instances of `Foo`).

Private

`private`, by contrast, limits access to instances of its class, and nothing else.

```
class Foo {  
    private fun something() {  
        println("Hello, world!")  
    }  
}  
  
val foo = Foo()  
  
foo.something()
```

This is the same as the previous snippet, except that `something()` is marked as `private`. This will not compile, as `something()` is not visible to code outside of `Foo`:

```
error: cannot access 'something': it is private in 'Foo'  
foo.something()
```

Private properties and functions can be referenced by other properties and functions within the same class:

```
class Foo {  
    private fun something() {  
        println("Hello, world!")  
    }  
  
    fun somethingElse() {  
        something()  
    }  
}
```

VISIBILITY AND SCOPE

```
}  
}  
  
fun main() {  
    val foo = Foo()  
  
    foo.somethingElse()  
}
```

(from "[private Members](#)" in the *Klassbook*)

This works, because we are calling `somethingElse()`, which is public (by default). While code outside of `Foo` cannot call `something()`, `somethingElse()` can call `something()`, as both `something()` and `somethingElse()` are part of `Foo`.

Protected

The protected visibility modifier works a lot like `private`, but it also allows subclasses to access the property or function:

```
open class Foo {  
    protected fun something() {  
        println("Hello, world!")  
    }  
}  
  
class Bar : Foo() {  
    fun somethingElse() {  
        something()  
    }  
}  
  
fun main() {  
    val notFoo = Bar()  
  
    notFoo.somethingElse()  
}
```

(from "[protected Members](#)" in the *Klassbook*)

This works, because `somethingElse()` is public (by default), and `Bar` can access the protected function inside of `Foo`. However, code outside of `Foo` and `Bar` cannot access that protected function, and so this fails:

```
open class Foo {
    protected fun something() {
        println("Hello, world!")
    }
}

class Bar : Foo() {
    fun somethingElse() {
        something()
    }
}

val notFoo = Bar()

notFoo.something()
```

What About Top-Level Functions and Properties?

So far, we have looked at how `private` and `protected` work with aspects of classes: properties and functions.

In the case of `protected`, that is the only place you can use that particular visibility. `protected` is defined as “accessible by this class and its subclasses”, so it only makes sense in the context of classes.

`private`, though, is not just available for aspects of classes. Classes themselves can be `private`. Properties and functions defined outside of classes can also be marked as `private`. In these cases, the determination of what can and cannot access the `private` items is based on the *file*:

- `private` top-level things are visible to other things in the same Kotlin source file
- `private` top-level things are inaccessible to other things in other Kotlin files

In a REPL scenario, everything in the REPL’s editor is in a single file, and so `private` has little meaning. `private` has a lot more meaning in a traditional software project, where you might have dozens or hundreds of source files. There, you can use `private` for access control:

- Anything left as `public` (the default) is accessible by other files
- Anything `protected` is accessible only by subclasses implemented in other files

- Anything marked as `private` is inaccessible in other files

What About Package-Private?

The closest thing that Java has to this sort of `private-to-the-file` rule is its default “package-private” visibility. By default, any class, method, or field is visible to other code in the same Java package, as determined by the package statement. Code in other packages cannot access that package-private code.

Kotlin does not offer package-private visibility. While packages can be useful for code organization, and may be important when [interoperating with Java code](#), Kotlin does not use them to control visibility.

What About Internal?

There is a fourth visibility option in Kotlin, called `internal`.

For most simple projects, `internal` is useless and is equivalent to being `public`.

Where `internal` comes into play is when your project is divided into a collection of some type of “modules”. For example, in an Android project, by default you have a single module (app), but you can elect to define additional modules. The precise definition of “module” depends entirely on where the Kotlin code is being used (e.g., an Android project) and is not defined by the language itself.

What *is* defined is what `internal` means: anything marked as `internal` is visible to code in the same module but is not visible to code in other modules.

This allows you to better define the API that a module exposes to other modules. Anything that you mark as `internal` is visible within your module, so you can use it without issue, but other modules should not be able to access it. Only the `public` and `protected` things will be visible to other modules (and `protected` is visible only to subclasses).

So, if you plan on publishing some form of Kotlin library, `internal` will be important. On a large project that you have divided into modules, `internal` can be useful. For small single-module projects, `internal` is pointless. And for REPLs, `internal` is just silly.

Scope

Visibility keywords control who can see what.

Scope controls where things exist. This has an indirect impact on visibility: if something does not exist, it cannot be accessed by anything.

However, scope more directly controls things like garbage collection, as scope helps to determine the lifetime of objects.

Global

Top-level definitions in a Kotlin source file are global in scope. Immutable globals — classes, `val`, etc. — will exist for the life of the process that is executing the Kotlin code. Mutable globals — such as `var` — have references that will live forever, but the objects that those references reference might come and go as the value of that reference changes.

So, in this code, both the `Foo` class and the `foo` variable are global and immutable:

```
class Foo {
    fun something() {
        println("Hello, world!")
    }
}

val foo = Foo()

foo.something()

var bar = Foo()

bar.something()

bar = Foo()

bar.something()
```

`bar` too is considered to be global. The difference is that `bar` is mutable. So while anything that `bar` points to cannot be garbage collected, once `bar` is pointing to something else (e.g., another instance of `Foo`), whatever `bar` *used* to point to can be garbage collected.

Instance

Properties defined inside of a class have “instance” scope. Each instance of the enclosing class will have its own set of instance properties, independent of any other instance of that same class.

```
class Foo(val count: Int)

fun main() {
    val instanceOne = Foo(1)
    val instanceTwo = Foo(2)

    println(instanceOne.count)
    println(instanceTwo.count)
}
```

(from "[Instance Scope](#)" in the *Klassbook*)

Here, each instance of `Foo` has its own `count` property with its own independent value.

Local

Any `val` or `var` within something smaller than a class is some form of local variable.

Local to Functions

A `val` or `var` directly inside of a function body is local to that function:

```
class Foo {
    fun something(message: String) {
        val length = message.length

        println("' $message' has $length characters")
    }
}

fun main() {
    Foo().something("this is a test")
}
```

(from "[Function-Local Scope](#)" in the *Klassbook*)

Here, `length` is considered to be local to the `something()` function. You would be

VISIBILITY AND SCOPE

unable to access `length` from other functions on `Foo`, let alone from outside of `Foo`.

Local to Blocks/Lambdas

A variable defined within a block or lambda (i.e., chunk of code in `{}`) will be local to that block or lambda:

```
class Foo {
    fun something(messages: List<String>) {
        messages.forEach { message ->
            val length = message.length

            println("' $message' has $length characters")
        }
    }
}

fun main() {
    Foo().something(listOf("this", "is", "a", "test"))
}
```

(from ["Block-Local Scope" in the Klassbook](#))

Now `length` is not local to `something()`, but instead is local to the lambda expression supplied to `forEach()`. You cannot access `length` from elsewhere within `something()`, let alone from other methods on `Foo` or from outside of `Foo`.

Abstract Classes and Interfaces

So far, we have looked at ordinary classes, sometimes referred to as “concrete” classes. You can have classes in a hierarchy, where an open class can be extended by subclasses. However, while a class can declare what is open to be overridden, a concrete class cannot stipulate something that *must* be implemented in a subclass. While a subclass can elect to override an open function, it does not have to do so.

Abstract classes and interfaces offer two ways to provide “contracts” between classes and their subclasses. These work similar to their Java counterparts, allowing you to declare functions with no default implementation, where the subclass is required by the compiler to provide an implementation.

The Objective: Contracts

Sometimes, we want to be able to get information about an object, but where the type of the object does not have a way of knowing that information.

For example, let’s go back to the class hierarchy that we saw in [the chapter on classes](#), adding in one more species:

```
open class Animal
class Frog : Animal()
class Axolotl : Animal()
class KomodoDragon : Animal()
```

We might want to know if an `Animal` has gills. Whether any given type of `Animal` has gills depends partly on the species and partly on circumstances:

- A [Komodo dragon](#) has no gills
- An axolotl has [external gills](#)
- A frog may have external gills briefly after hatching, but they get absorbed back into the body shortly thereafter

As a result, we cannot implement a `hasGills()` function or `hasGills` property on `Animal` very easily... unless we use abstract classes or interfaces. Then, we can ask an `Animal` if it has gills, but the actual implementation would be delegated to `Frog`, `Axolotl`, or `KomodoDragon`.

Abstract Classes

An abstract class has the `abstract` keyword:

```
abstract class Animal {
    abstract fun hasGills(): Boolean
}
```

An abstract class is automatically open as well — we do not need the `open` keyword to be able to create subclasses.

As with Java, we cannot create instances of an abstract class. If we try to do so:

```
abstract class Animal {
    abstract fun hasGills(): Boolean
}

val critter = Animal()
```

...we fail with a compile error:

```
error: cannot create an instance of an abstract class
val critter = Animal()
               ^
```

Abstract Members

We can then start defining members of the class that also have the `abstract` keyword. Mostly, that will be abstract functions:

ABSTRACT CLASSES AND INTERFACES

```
abstract class Animal {
    abstract fun hasGills(): Boolean
}
```

An abstract class can have ordinary functions as well — you are not limited to just ones with the `abstract` keyword. Ordinary functions can themselves be open or not, as you see fit:

```
abstract class Animal {
    abstract fun hasGills(): Boolean

    open fun displayName(): String = "Animal"

    fun description() = "${displayName(): hasGills = ${hasGills()}"
}
```

It is also possible to create abstract properties, though we will hold off on this until [much later in the book](#).

Concrete and Abstract Subclasses

A subclass of an abstract class either:

- Needs to be abstract itself, or
- Needs to have implementations of all members defined as abstract that it inherited

So, for example, this does not work:

```
abstract class Animal {
    abstract fun hasGills(): Boolean

    fun description() = "${displayName(): hasGills = ${hasGills()}"
}

class Frog : Animal()

class Axolotl : Animal()

class KomodoDragon : Animal()
```

We get a compile error for each of the `Animal` subclasses:

ABSTRACT CLASSES AND INTERFACES

```
error: class 'Frog' is not abstract and does not implement abstract base class
member public abstract fun hasGills(): Boolean defined in Test.Animal
class Frog : Animal()
^
```

Since `hasGills()` is abstract, the subclasses need to be either abstract or provide an implementation for `hasGills()`:

```
abstract class Animal {
    abstract fun hasGills(): Boolean

    fun description() = "hasGills = ${hasGills()}"
}

class Frog(val hasGillsRightNow: Boolean) : Animal() {
    override fun hasGills() = hasGillsRightNow
}

class Axolotl : Animal() {
    override fun hasGills() = true
}

class KomodoDragon : Animal() {
    override fun hasGills() = false
}

fun main() {
    println(Axolotl().description())
    println(KomodoDragon().description())
    println(Frog(true).description())
}
```

(from "[Abstract Classes](#)" in the *Klassbook*)

Exactly *how* the subclass implements `hasGills()` is up to that subclass. `Axolotl` and `KomodoDragon` have hard-coded responses, while `Frog` has its `hasGills()` depend upon a constructor parameter.

Interfaces

In the first decade or so of Java, there was a clear distinction between abstract classes and interfaces:

- Abstract classes could have concrete methods and fields, along with abstract methods delegated to subclasses to implement

ABSTRACT CLASSES AND INTERFACES

- Interfaces were purely a set of abstract methods, used for defining a contract

Java 8 started to blur the lines, allowing interfaces to offer “default” methods (concrete methods designed to be overridden as needed).

Kotlin offers its own take on interfaces, which largely mirrors Java 8’s approach.

Basic Definition

Declaring an interface in Kotlin is similar to how you declare them in Java: replace class with interface and (usually) have 1+ members (e.g., functions). You do not need the abstract keyword for functions without bodies, as it is assumed that those are abstract by default:

```
interface Aquatic {  
    fun hasGills(): Boolean  
}
```

However, there is no requirement for an interface to have *any* abstract members, though, so this is legal:

```
interface Aquatic {  
    fun hasGills() = false  
}
```

Basic Usage

In Java, having a class implement an interface starts with the `implements` keyword:

```
class Animal implements Aquatic {  
    // TODO stuff goes here  
}
```

In Kotlin, interfaces are just listed alongside any superclasses in a comma-delimited list after the `:` in the class declaration:

```
interface Aquatic {  
    fun hasGills(): Boolean  
}  
  
open class Animal  
  
abstract class AquaticAnimal : Animal(), Aquatic {
```


ABSTRACT CLASSES AND INTERFACES

```
fun description() = "hasGills = ${hasGills()}"
}

class Frog(val hasGillsRightNow: Boolean) : AquaticAnimal() {
    override fun hasGills() = hasGillsRightNow
}

class Axolotl : AquaticAnimal() {
    override fun hasGills() = true
}

fun main() {
    println(Axolotl().description())
    println(Frog(true).description())
}
```

(from "[Interfaces](#)" in the *Klassbook*)

Here, we have:

- the Aquatic interface, to be used for animals that live primarily in the water
- a base Animal class that is open for subclasses
- an AquaticAnimal that extends Animal and also implements the Aquatic interface
- Frog and Axolotl extending AquaticAnimal and implementing the abstract hasGills() function

Note that AquaticAnimal needs to be declared as abstract, as while it “implements” Aquatic, it does not satisfy the Aquatic contract by implementing hasGills(). Since hasGills() is being deferred to subclasses to implement, we need to make AquaticAnimal be an abstract class. We do not need to repeat the abstract fun hasGills(): Boolean declaration, though — the one that we get from Aquatic is sufficient.

Also note that in the list of classes and interfaces as part of the AquaticAnimal declaration, classes use constructor notation (parentheses and, if applicable, parameters to pass to the superclass’ constructor). Interfaces, though, are just the name of the interface, without parentheses.

Using Multiple Interfaces

Kotlin does not allow a class to extend from multiple superclasses. However, Kotlin does allow a class to implement multiple interfaces, just by adding them to the

ABSTRACT CLASSES AND INTERFACES

declaration list:

```
interface Aquatic {
    fun hasGills(): Boolean
}

interface ReproductionMode {
    fun isOviparous(): Boolean    // i.e., does it lay eggs?
}

open class Animal

abstract class AquaticAnimal : Animal(), Aquatic, ReproductionMode {
    fun description() = "hasGills = ${hasGills()}, isOviparous() = ${isOviparous()}"
}

class Frog(val hasGillsRightNow: Boolean) : AquaticAnimal() {
    override fun hasGills() = hasGillsRightNow
    override fun isOviparous() = true
}

class Axolotl : AquaticAnimal() {
    override fun hasGills() = true
    override fun isOviparous() = true
}

class Orca : AquaticAnimal() {
    override fun hasGills() = false
    override fun isOviparous() = false
}

class KomodoDragon : Animal(), ReproductionMode {
    override fun isOviparous() = true
}

fun main() {
    println(Axolotl().description())
    println(Frog(true).description())
    println(Orca().description())
}
```

(from "[Multiple Interfaces](#)" in the *Klassbook*)

Here, we have `AquaticAnimal` implement both `Aquatic` and `ReproductionMode`.

Of course, since all animals reproduce, we could have `Animal` implement `ReproductionMode` and simplify this a bit, though it will mean that by default `Animal` would need to be abstract:

```
interface Aquatic {
    fun hasGills(): Boolean
}

interface ReproductionMode {
    fun isOviparous(): Boolean    // i.e., does it lay eggs?
}
```

ABSTRACT CLASSES AND INTERFACES

```
}

abstract class Animal : ReproductionMode

abstract class AquaticAnimal : Animal(), Aquatic

class Frog(val hasGillsRightNow: Boolean) : AquaticAnimal() {
    override fun hasGills() = hasGillsRightNow
    override fun isOviparous() = true
}

class Axolotl : AquaticAnimal() {
    override fun hasGills() = true
    override fun isOviparous() = true
}

class Orca : AquaticAnimal() {
    override fun hasGills() = false
    override fun isOviparous() = false
}

class KomodoDragon : Animal() {
    override fun isOviparous() = true
}
```

Bear in mind that any abstract class is also open, and so we could default the implementation of interface functions in superclasses, to reduce redundancy a bit:

```
interface Aquatic {
    fun hasGills(): Boolean
}

interface ReproductionMode {
    fun isOviparous(): Boolean // i.e., does it lay eggs?
}

abstract class Animal : ReproductionMode {
    override fun isOviparous() = false
}

abstract class AquaticAnimal : Animal(), Aquatic {
    override fun hasGills() = true
}

class Frog(val hasGillsRightNow: Boolean) : AquaticAnimal() {
    override fun hasGills() = hasGillsRightNow
    override fun isOviparous() = true
}
```

ABSTRACT CLASSES AND INTERFACES

```
}  
  
class Axolotl : AquaticAnimal() {  
    override fun isOviparous() = true  
}  
  
class Orca : AquaticAnimal() {  
    override fun hasGills() = false  
}  
  
class KomodoDragon : Animal() {  
    override fun isOviparous() = true  
}
```

Now Frog, Axolotl, Orca, and KomodoDragon only need to override those functions where their needs differ from the default supplied by Animal or AquaticAnimal.

Concrete Functions

Not only can interfaces have functions marked as abstract, but they can have regular concrete functions as well. As this sample shows, interfaces do not necessarily need any abstract functions at all:

```
interface Aquatic {  
    fun hasGills() = true  
}  
  
interface ReproductionMode {  
    fun isOviparous() = false    // i.e., does it lay eggs?  
}  
  
abstract class Animal : ReproductionMode  
  
abstract class AquaticAnimal : Animal(), Aquatic  
  
class Frog(val hasGillsRightNow: Boolean) : AquaticAnimal() {  
    override fun hasGills() = hasGillsRightNow  
    override fun isOviparous() = true  
}  
  
class Axolotl : AquaticAnimal() {  
    override fun isOviparous() = true  
}  
  
class Orca : AquaticAnimal() {  
    override fun hasGills() = false
```

```
}  
  
class KomodoDragon : Animal() {  
    override fun isOviparous() = true  
}
```

Here, rather than putting the default implementations of `hasGills()` and `isOviparous()` on the `Animal` and `AquaticAnimal` classes, we put them directly on the interfaces themselves.

Which Do You Use?

On the surface, it seems like interfaces can do nearly everything that abstract classes can do, and you gain the flexibility of being able to use more than one interface on a class.

So, why would one ever use an abstract class? Or, to put it another way, in what scenarios would you use abstract classes, and in what scenarios would you use interfaces?

State Management

The biggest thing that interfaces lack is any form of state management. They cannot have concrete properties, the way abstract classes (and regular classes) can.

So, if you want to have something that manages state but requires implementations to fulfill some contract, an abstract class is the better choice.

In principle, you could use interfaces and simply force implementations to do some of the state management themselves. For example, you could have abstract `getFoo()` and `setFoo()` functions declared in the interface, plus have concrete code in the interface rely upon those functions. In principle, this can work. However, the interface has no control over exactly how that state is maintained:

- Is it one “foo” per instance of the class?
- Is it one “foo” per process (e.g., a top-level `var`)?
- Is it one “foo” per some sort of context (e.g., per user in a Web app)?

If the interface truly does not care and could handle any of those scenarios, fine. Otherwise, an abstract class would be a better choice, so the concrete code *knows* exactly how the state management is being handled.

On the other hand, if you have some code that is more of a “mixin” and does not need any particular state, then interfaces are fine.

Consumer of Contracts

Another subtle limitation is that interfaces cannot have protected functions. The abstract functions on an interface are intrinsically public, though their concrete functions can be private if desired.

Hence, if you want to require subclasses to implement some functionality, but that functionality should not be accessible to arbitrary other objects, an abstract class with protected abstract functions is the best choice.

If, on the other hand, all of the abstract functions are fine for other objects to call, then interfaces are fine.

I Can Haz final?

Interfaces can inherit from other interfaces, using the same notation as one uses to inherit with classes:

```
interface Base {
    fun foo()
}

interface Sub : Base {
    fun bar()

    fun goo() {
        // do something
    }
}
```

However, an interface cannot mark an inherited abstract method as `final`, even if it provides an implementation. So, this is fine:

```
interface Base {
    fun foo()
}

interface Sub : Base {
    fun bar()
```

ABSTRACT CLASSES AND INTERFACES

```
fun goo() {  
    // do something  
}  
  
override fun foo() {  
    // this supplies an implementation  
}  
}
```

...but this is not:

```
interface Base {  
    fun foo()  
}  
  
interface Sub : Base {  
    fun bar()  
  
    fun goo() {  
        // do something  
    }  
  
    override final fun foo() {  
        // this supplies an implementation  
    }  
}
```

Instead, we get a compile error:

```
error: modifier 'final' is not applicable inside 'interface'  
    override final fun foo() {  
        ^
```

Abstract classes, like regular classes, can override inherited methods and declare those overridden methods as `final`, to prevent further subclasses from overriding them.

As a result, anything in an interface hierarchy is permanently open, until you start implementing the interfaces in classes. If that is a problem — if you have some function that you really want to mark as `final` — use abstract classes, not interfaces.

Data Class

One of the reasons why Kotlin has taken off in popularity — particularly for Android app development — is that Java can be tedious to write.

One way that developers have tried to reduce that tedium is through annotation processors that generate Java code for you. A classic example of that is Google's [AutoValue](#). AutoValue is for immutable objects, ones whose fields have getters but no setters. In particular, AutoValue can code-generate things like an `equals()` method that takes all of the fields into account.

In Kotlin, a data class offers the same basic feature set as AutoValue, with simpler syntax, because it is “baked into” the language itself.

How You Declare It

All you do is add the `data` keyword to the class declaration:

```
data class Animal(val species: String, val ageInYears: Float)
```

The class needs to have 1+ parameters in its constructor, and those parameters need to be declared as `val` or `var...` and typically you will use `val`.

And that's pretty much it. You are welcome to have whatever functions you need on the class, but nothing else is required.

What You Gain

OK, so what does that `data` keyword give us?

Standard Java Methods

Kotlin will automatically supply implementations of:

- `equals()`, which compares each of the properties to see if they are equal
- `hashCode()`, which generates an `Int` to be used in places like `HashSet` and `HashMap`
- `toString()`, mostly for debugging purposes, showing the values of each of the properties

For Kotlin/JVM, these methods are the standard Java `Object` methods that ideally get overridden on most classes... yet do not, because we skip them if we think that we do not need them.

This is a subset of the Java equivalent of the `Animal` data class shown above:

```
public final class Animal {
    @NotNull
    private final String species;
    private final float ageInYears;

    @NotNull
    public final String getSpecies() {
        return this.species;
    }

    public final float getAgeInYears() {
        return this.ageInYears;
    }

    public Animal(@NotNull String species, float ageInYears) {
        this.species = species;
        this.ageInYears = ageInYears;
    }

    public String toString() {
        return "Animal(species=" + this.species + ", ageInYears=" + this.ageInYears +
        ")";
    }

    public int hashCode() {
        return (this.species != null ? this.species.hashCode() : 0) * 31 +
        Float.floatToIntBits(this.ageInYears);
    }
}
```

DATA CLASS

```
public boolean equals(Object var1) {
    if (this != var1) {
        if (var1 instanceof Animal) {
            Animal var2 = (Animal)var1;

            if (this.species.equals(var2.species) && Float.compare(this.ageInYears,
var2.ageInYears) == 0) {
                return true;
            }
        }

        return false;
    } else {
        return true;
    }
}
}
```

(we will see where this code came from [later in the book](#), though it has been simplified here for the purposes of this chapter)

Kotlin code-generates more than this, such as the `copy()` function described in the next section. But, this gives you an idea of what you are getting for `toString()`, `hashCode()`, and `equals()`.

Note, though, that Kotlin will only generate these methods for you if they are needed:

- If you implement them yourself in your data class, Kotlin will assume that you know what you are doing
- If your data class extends from some other class, Kotlin will skip any of those functions that the class implements and marks as `final`

`copy()`

Kotlin also generates a `copy()` function. As the name suggests, this creates a copy of an instance of your data class. However, what the function really does is accept all of the properties as function parameters, defaulted to the current values from the instance. This allows you to selectively override values in the copy, which is a great place to use named parameters:

```
data class Animal(val species: String, val ageInYears: Float)

fun main() {
```

DATA CLASS

```
val critter = Animal("frog", 3.14F)
val youngerCriticter = critter.copy(ageInYears = 0.1F)

println(youngerCriticter)
}
```

(from ["Data Class" in the Klassbook](#))

This results in:

```
Animal(species=frog, ageInYears=0.1)
```

So, our second `Animal` copied the species but has a different `ageInYears`, courtesy of the value that we provided to `copy()`.

Data classes are very useful in implementing [immutable objects](#). The `copy()` function then becomes a key way to create a replacement edition of an object: rather than changing the existing instance, you create a copy with all of the current information except what needs to change.

What You Lose

You cannot create subclasses of data classes. A data class cannot be marked with `open` to allow it to be extended.

If subclasses were allowed, it is possible that subclasses would change the class definition in ways that might make the code-generated `equals()` and other methods be invalid. Hence, at least right now, Kotlin prohibits subclasses of data classes.

A side-effect of the no-subclasses limitation is that data classes cannot be abstract.

Data Classes with Other Properties

You are welcome to have other properties in your data classes, beyond those in the constructor:

```
data class Animal(val species: String, val ageInYears: Float) {
    var isFriendly = true
    var isHungry = true
    val isCommonlySeenFlyingInTornadoes = false
}
```

DATA CLASS

However, code-generated functions, like `equals()` and `copy()`, will ignore these properties. Those functions only incorporate the properties defined in the primary constructor.

Sometimes, this can be a feature: you might want some properties to be ignored for `equals()` and `hashCode()`. Sometimes, this can be a bug: you might not realize that `copy()` only copies a subset of your properties.

The object Keyword

Most likely, there have been developers over the years who have wondered why object-oriented programming languages do not have any sort of object keyword.

The designers of Kotlin may have been among those developers, as they have given Kotlin such an object keyword. This gets used in a few different places in Kotlin syntax

Singletons

One way to declare singletons in Kotlin is to use a top-level `val` declaration:

```
class Transmogriifier {
    fun transmogrify() {
        println("Presto, change-o!")
    }
}

val QUASI_SINGLETON = Transmogriifier()

fun main() {
    QUASI_SINGLETON.transmogriify()
}
```

(from "[Manual Singletons](#)" in the *Klassbook*)

In truth, though, we can have as many instances of `Transmogriifier` as we want. `QUASI_SINGLETON` is merely a global instance of `Transmogriifier`. We cannot prevent others from making their own `Transmogriifier` instances, as then we could not create the quasi-singleton. For example, we cannot make the constructor be private, like this:

THE OBJECT KEYWORD

```
class Transmogrifier private constructor() {
    fun transmogrify() {
        // TODO
    }
}

val QUASI_SINGLETON = Transmogrifier()

QUASI_SINGLETON.transmogrify()
```

as that results in a compile error:

```
error: cannot access '<init>': it is private in 'Transmogrifier'
val QUASI_SINGLETON = Transmogrifier()
                        ^
```

To create a true singleton, all we need to do is:

- Replace the class with object
- Call functions on that object based on its name, rather than creating some instance ourselves

```
object Transmogrifier {
    fun transmogrify() {
        println("Presto, change-o!")
    }
}

fun main() {
    Transmogrifier.transmogrify()
}
```

(from ["Object Singletons" in the Klassbook](#))

An object declared this way can have properties and functions like a regular class. It cannot have a constructor, which makes sense, since there will only ever be one instance. However, if it extends classes, it can (and must) call the superclass constructors:

```
open class Base {
    // TODO stuff here
}

object Transmogrifier : Base() {
    fun transmogrify() {
        println("Presto, change-o!")
    }
}
```

THE OBJECT KEYWORD

```
}  
}  
  
fun main() {  
    Transmogrifier.transmogrify()  
}
```

(from ["Objects and Superclasses" in the Klassbook](#))

As with singletons in any programming environment, though, the ones that we declare via `object` will live for the life of our process, so be careful that you are not creating a memory leak by having an object hold onto more and more stuff over time.

Companion Objects

Another use of the `object` keyword is for a companion of a class. A companion is a singleton, associated with the class, whose functions somewhat fill the role that static methods do in Java. In fact, if you set up those functions the right way, they can be [called from Java as static methods](#).

Declaring and Using a Companion Object

A companion object is simply an object, nested inside of a class, prefixed with the `companion object` keyword:

```
class Thingy {  
    companion object {  
        fun doSomething() {  
            println("Ummm... is this something?")  
        }  
    }  
  
    // TODO add other properties and functions  
}  
  
fun main() {  
    Thingy.doSomething()  
}
```

(from ["Companion Objects" in the Klassbook](#))

Then, other parties can call the companion object's functions as if they were "static" functions on the enclosing class (e.g., `Thingy.doSomething()`).

Why Bother?

It is entirely possible that you are unimpressed by this. After all, we can have top-level functions in Kotlin:

```
fun doSomething() {  
    // TODO  
}  
  
doSomething()
```

However, companion objects have one key advantage: they have access to private functions and properties of their enclosing class. So, this works:

```
class Thingy {  
    private val count = 1  
  
    companion object {  
        fun doSomething(thingy: Thingy) {  
            println(thingy.count)  
        }  
    }  
  
    // TODO add other properties and functions  
}  
  
fun main() {  
    Thingy.doSomething(Thingy())  
}
```

(from "[Companion Objects and Private Members](#)" in the *Klassbook*)

Even though `count` is private, `doSomething()` can still reference it, since `doSomething()` is a part of the `Thingy` implementation.

By contrast, a top-level function cannot reference private functions and properties, so this fails with a compile error:

```
class Thingy {  
    private val count = 1  
}  
  
fun doSomething(thingy: Thingy) {  
    println(thingy.count)  
}
```

Naming a Companion Object

Sometimes, you will see a companion object have a name, though probably not one named after a [companion](#), as we have here:

```
class Thingy {
    private val count = 1

    companion object AmyPond {
        fun doSomething(thingy: Thingy) {
            println(thingy.count)
        }
    }

    // TODO add other properties and functions
}

fun main() {
    Thingy.doSomething(Thingy())
}
```

(from "[Named Companion Objects](#)" in the *Klassbook*)

This has limited impact on your Kotlin code — you still call functions on the class as before. It does have an impact if you are going to try using the companion object functions from Java, as we will explore [later in the book](#).

Nested Objects

A companion object is just a specialized case of nested objects, where we have “singletons” declared inside of a class. We will examine nested objects in general more [in the next chapter](#).

Object Expressions

Frequently, we need to pass an instance of an object to some function, where we need a tailored instance of the desired class, with custom functionality.

In Java, this is where the “anonymous inner class” comes into play:

```
void clickyClickyClicky(View view) {
    view.setOnClickListener(new OnClickListener() {
        @Override
```

```
public void onClick(View v) {  
    // TODO something useful  
}  
});  
}
```

This sort of code will be familiar to long-time Android developers. To find out when this UI element (a `View`) gets clicked, we can call `setOnClickListener()`, providing an implementation of a `OnClickListener` interface. Rather than create a standalone class, though, we use anonymous inner class syntax to create a class and a single instance of that class.

Kotlin's equivalent of that sort of syntax is the “object expression”.

Basic Declaration

The Kotlin equivalent of the `View` and `OnClickListener` from above might look like this:

```
interface OnClickListener {  
    fun onClick(v: View)  
}  
  
class View {  
    fun setOnClickListener(listener: OnClickListener) {  
        // TODO something with this  
    }  
}
```

(technically, in Android, `OnClickListener` is a nested interface inside of `View` — we will explore that pattern [in an upcoming chapter](#))

We have the `View` class and an `OnClickListener` interface, along with a `setOnClickListener()` function.

When we want to call `setOnClickListener()`, we could create a standalone class that implements the `OnClickListener` interface. Or, we could use an object expression instead:

```
fun clickyClickyClicky(view: View) {  
    view.setOnClickListener(object : OnClickListener {  
        override fun onClick(v: View) {  
            // TODO something useful  
        }  
    })  
}
```

```
    }  
  })  
}
```

The object keyword indicates that we are creating a single object, and the `:` `OnClickListener` indicates that this object will implement the `OnClickListener` interface. From there, it is no different than any other interface implementation: we override the `onClick()` function and do whatever it is that we want to do.

Constructors and Parameters

Object expressions can extend other classes and implement interfaces. The notation is pretty much the same as when you have a class extend other classes and implement interfaces. The biggest difference is that you use the object keyword in place of the class name.

In the above example, `OnClickListener` is an interface. As a result, our object expression could just have the interface name after the colon (`: OnClickListener`) to implement it, because interfaces have no constructors. Classes do, so if `OnClickListener` were an abstract class, you would need to invoke a constructor in your object expression:

```
abstract class OnClickListener {  
    abstract fun onClick(v: View)  
}  
  
class View {  
    fun setOnClickListener(listener: OnClickListener) {  
        // TODO something with this  
    }  
}  
  
fun clickyClickyClicky(view: View) {  
    view.setOnClickListener(object : OnClickListener() {  
        override fun onClick(v: View) {  
            // TODO something useful  
        }  
    })  
}
```

Access Rules

Your object expression has access to anything that is available to other Kotlin code in

THE OBJECT KEYWORD

the same scope. So, for example, if there is a local variable in a function where the object expression is declared, the local variable is accessible by the object expression's own code:

```
fun clickyClickyClicky(view: View) {
    var clicks = 0

    view.setOnClickListener(object : OnClickListener() {
        override fun onClick(v: View) {
            clicks++
            println(clicks)
        }
    })
}
```

In Java, this requires the `final` keyword to work, and you would not be allowed to modify the value the way we are doing here. Kotlin is more flexible: you do not need any keywords and you have normal access to the variable.

The SAM Scenario

On occasion, you will find Kotlin code that seems to be akin to an object expression but just has a lambda for its implementation:

```
val listener = OnClickListener { v: View -> ... }
```

(where `...` is some useful Kotlin code)

This is the single-abstract-method pattern. We will examine it in greater detail [later in the book](#), because as it turns out, this only works if the interface being used (`OnClickListener`) is defined in Java, not in Kotlin.

Nested Objects and Classes

In [the previous chapter](#), we saw the use of a companion object. This is a special scenario for nested objects, where we have object values as properties of a class.

Not only can a class have nested objects, but it can have nested classes, where we have one class defined inside of another one.

Nested Objects

We saw companion object in the previous chapter:

```
class Thingy {
  companion object {
    fun doSomething() {
      println("Ummm... is this something?")
    }
  }

  // TODO add other properties and functions
}

fun main() {
  Thingy.doSomething()
}
```

(from "[Companion Objects](#)" in the *Klassbook*)

The value of the companion keyword is that functions on the companion object can be called just like you might call static functions in Java:

```
Thingy.doSomething()
```

However, you can use the `object` keyword for nested objects without necessarily using `companion`.

Named Objects

One option is to give the object a name, akin to how you give a class a name:

```
class Thingy {
    object Somethingifier {
        fun doSomething() {
            println("Ummm... is this something?")
        }
    }

    // TODO add other properties and functions
}

fun main() {
    Thingy.Somethingifier.doSomething()
}
```

(from "[Named Nested Objects](#)" in the *Klassbook*)

Then, you reference it via dot notation (`Thingy.Somethingifier.doSomething()`).

Object Properties

A more typical approach, though, is simply to assign the object to a property, using an object expression as the property initializer:

```
interface Somethingifier {
    abstract fun doSomething()
}

class Thingy {
    val somethingifier = object : Somethingifier {
        override fun doSomething() {
            println("Ummm... is this something?")
        }
    }

    // TODO add other properties and functions
}

fun main() {
```

NESTED OBJECTS AND CLASSES

```
val thingy = Thingy()

thingy.somethingifier.doSomething()
}
```

(from ["Object Properties" in the Klassbook](#))

You can then reference it as you would any other property (thingy.somethingifier.doSomething()).

Nested Classes

Java supports nested classes:

```
class Foo {
    class Bar {
        // TODO good stuff
    }

    Bar bellyUp() {
        return new Bar();
    }

    // TODO even more good stuff
}

Foo.Bar bar = new Foo().bellyUp();
```

Similarly, Kotlin supports nested classes. But, as with many things in Kotlin, there are some differences between Java's approach and Kotlin's.

Simple Nested Classes

You can have a class inside of a class easily enough in Kotlin:

```
class Foo {
    class Bar {
        fun whatever() {
            println("You were expecting something profound?")
        }

        // TODO add good stuff
    }

    // TODO add even more good stuff
}
```


NESTED OBJECTS AND CLASSES

```
}  
  
fun main() {  
    val bar = Foo.Bar()  
  
    bar.whatever()  
}
```

(from "[Nested Classes](#)" in the [Klassbook](#))

However, this is not quite the same as the Java example above. The default behavior in Kotlin is more akin to Java's static class:

```
class Foo {  
    static class Bar {  
        // TODO good stuff  
    }  
  
    // TODO even more good stuff  
}  
  
Foo.Bar bar = new Foo.Bar();
```

An instance of Bar is not tied to any instance of Foo. In many respects, the nested class is only connected to its outer class by name — you might just as easily have:

```
class Bar {  
    // TODO good stuff  
}  
  
class Foo {  
    // TODO even more good stuff  
}  
  
val bar = Bar()
```

The biggest value of a nested class, over a completely independent class, is that a nested class has access to private values that are in an eligible scope. That would include values held by an object inside of the outer class:

```
class Foo {  
    private object Data {  
        const val VALUE = 1  
    }  
  
    class Bar {
```

NESTED OBJECTS AND CLASSES

```
fun value() = Foo.Data.VALUE

    // TODO add good stuff
}

// TODO add even more good stuff
}

fun main() {
    val bar = Foo.Bar()

    println(bar.value())
}
```

(from "[Nested Classes and Private Access](#)" in the *Klassbook*)

Here, Bar can access Foo.Data, even though that object is private, since Bar is nested inside of Foo. Classes and objects outside of Foo have no direct access to Data.

Inner Classes

Let's go back to the original Java snippet:

```
class Foo {
    class Bar {
        // TODO good stuff
    }

    Bar bellyUp() {
        return new Bar();
    }

    // TODO even more good stuff
}

Foo.Bar bar = new Foo().bellyUp();
```

In Java, by default — as in the case with Foo and Bar above — the nested class is an “inner” class. An instance of Bar has access to everything inside of an enclosing instance of Foo.

```
class Foo {
    int count = 1;

    class Bar {
        int getCount() {
```

NESTED OBJECTS AND CLASSES

```
        return count;
    }
}

Foo.Bar bar = new Foo().bellyUp();
int theCount = bar.getCount();
```

Our instance of Bar has the ability to “reach into” an enclosing instance of Foo and access its members, including fields and methods. In this case, Bar accesses the count field of Foo.

The equivalent in Kotlin requires you to use the inner keyword when declaring the inner class:

```
class Foo {
    val count = 1

    inner class Bar {
        fun count() = count
    }
}

fun main() {
    val foo = Foo()
    val bar = foo.Bar()

    println(bar.count())
}
```

(from "[Inner Classes](#)" in the *Klassbook*)

An instance of Bar can reference the count inside of the enclosing instance of Foo.

In both Java and Kotlin, only an *instance* of the outer class can create an instance of the inner class. In our case, we need an instance of Foo to be able to ask it to create for us an instance of Bar. In the case of Kotlin, because creating new objects does not require a keyword, we can just invoke the Bar() constructor on an instance of Foo. In Java, typically we use some method on the outer class to create instances of the inner class for us (e.g., bellyUp()).

Also, to make this work, the instance of the inner class has an *implicit* reference to the instance of the outer class. We do not have an explicit field for it, but it is there. This causes garbage collection problems, as developers sometimes forget that a Bar has a hidden reference to a Foo, and therefore the Foo cannot be garbage-collected

while we are holding onto a Bar.

Which this is this?

Inner classes raise a thorny problem: when we use `this`, what instance is it referring to?

By default, `this` refers to the instance of the inner class, so this code prints `class Bar` or `class Foo$Bar`, depending on what Kotlin platform you run on:

```
class Foo {
    inner class Bar {
        fun that() = this
    }
}

fun main() {
    val foo = Foo()
    val bar = foo.Bar()

    println(bar.that()::class)
}
```

(from "[Inner Classes and this](#)" in the *Klassbook*)

(`$` notation indicates an inner class relationship, and `Foo$Bar` means “the Bar class that is an inner class of Foo”, but this will only be seen in Kotlin/JVM, not the Kotlin/JS that *Klassbook* uses)

In other words, `this` by default refers to the instance of the inner class (Bar, in this case).

Occasionally, though, we need to specifically get a reference to the instance of the outer class. Android Java developers run into this frequently:

- In some Activity, you define an anonymous inner class to use for a callback, such as a `View.OnClickListener`
- In a method in that anonymous inner class, you need a reference to the Activity
- You try using `this`, and you get compile errors, saying that `this` is not an Activity (or Context or other superclass)

Java has syntax to reaching the outer class instance:

NESTED OBJECTS AND CLASSES

- `this` refers to the inner class instance
- `OuterClass.this` refers to the outer class instance that created the current inner class instance that a plain `this` refers to

Kotlin has the same concept, with different syntax:

- `this` refers to the inner class instance
- `this@OuterClass` refers to the outer class instance that created the current inner class instance that a plain `this` refers to

```
class Foo {
    val count = 1

    inner class Bar {
        fun count() = count

        fun that() = this

        fun theOuterThis() = this@Foo
    }
}

fun main() {
    val foo = Foo()
    val bar = foo.Bar()

    println(bar.that()::class)
    println(bar.theOuterThis()::class)
}
```

(from "[Inner Classes and Outer this](#)" in the *Klassbook*)

This snippet prints something like:

```
class Bar
class Foo
```

`this@Foo` returns a `Foo`, specifically the instance of `Foo` that created the `Bar` instance on which we called `theOuterThis()`.

Nested Interfaces and Abstract Classes

Not only can you nest classes and objects inside of classes, but you can nest interfaces as well:

NESTED OBJECTS AND CLASSES

```
class Thingy {
    interface Transmogrifier {
        abstract fun transmogrify(): Thingy
    }

    // TODO good stuff here
}

val moggle = object : Thingy.Transmogrifier {
    override fun transmogrify() = Thingy()
}

val thingy = moggle.transmogrify()
```

Or, if you prefer, you can nest abstract classes:

```
class Thingy {
    abstract class Transmogrifier {
        abstract fun transmogrify(): Thingy
    }

    // TODO good stuff here
}

val moggle = object : Thingy.Transmogrifier() {
    override fun transmogrify() = Thingy()
}

val thingy = moggle.transmogrify()
```

Naming works the same as with nested classes and named objects: you reference the nested interface or abstract class based on the outer class name (e.g., `Thingy.Transmogrifier`).

Enums and Sealed Classes

A boolean is a way of modeling a fixed set of states, where there are only two possible states.

Frequently, though, we find ourselves needing to model a fixed set of states that has more than two entries. For example, even with something that is nominally a boolean, we might have “true”, “false”, and “undefined”. The latter state would be for cases where we have not yet gone through any code that positively sets the state to “true” or “false”. For example, for some sort of user preference, perhaps the user has not visited the “settings” screen to state their preference just yet, so their preference is “undefined”.

The more you look, the more you find things that might be modeled this way:

- The error responses from a server API
- The steps in a wizard-style “signup” set of screens
- The modes of transportation that a Person object might be taking (walking? biking? driving? flying? skiing? jet-skiing?)
- And so on

For these cases, Kotlin offers two programming constructs: enumerations (“enums”) and sealed classes. In this chapter, we will explore each of these.

Enums

Some programming languages — particularly those that grew out of C — offer “enums” as first-class constructs. So, Java has an enum:


```
enum ServerError {
    INVALID_INPUT,
    OUT_OF_STORAGE_SPACE,
    USER_NOT_FOUND,
    CLIENT_NOT_FOUND,
    SERVER_NOT_FOUND,
    WHOEVER_YOU_THINK_YOU_ARE_TALKING_TO_NOT_FOUND
}
```

Kotlin's enum option resembles that of Java.

Basic Syntax and Usage

The first difference between Kotlin and Java is in the declaration. In Java, enum exists as a standalone keyword. In Kotlin, the enum decorates class, much like how data is a type of class:

```
enum class HungerState {
    NOT_HUNGRY,
    SOMETIMES_HUNGRY,
    ME_ALWAYS_HUNGRY,
    RAVENOUS,
    YOU_LOOK_LIKE_FOOD
}
```

You can then refer to enum values using standard dot notation, such as `HungerState.ME_ALWAYS_HUNGRY`.

Since this is a type of class, you use an enum in Kotlin the same way that you would any other class, such as in a constructor parameter:

```
enum class HungerState {
    NOT_HUNGRY,
    SOMETIMES_HUNGRY,
    ME_ALWAYS_HUNGRY,
    RAVENOUS,
    YOU_LOOK_LIKE_FOOD
}

data class Animal(
    val species: String,
    val ageInYears: Float,
    val hungry: HungerState = HungerState.SOMETIMES_HUNGRY
) {
```

```
var isFriendly = true
val isCommonlySeenFlyingInTornadoes = false
}
```

Class Features

Since a Kotlin enum is a class, you can use many class features in an enum.

Constructors

The most commonly seen of these is the constructor, used to provide values to define an individual enum constant:

```
enum class HttpResponse(val code: Int) {
    OK(200),
    MOVED_PERMANENTLY(301),
    NOT_MODIFIED(304),
    UNAUTHORIZED(401),
    NOT_FOUND(404),
    INTERNAL_SERVER_ERROR(501)
}

fun main() {
    println(HttpResponse.OK.code)
}
```

(from ["enum Class" in the Klassbook](#))

Here, we use a constructor to associate a code value with each `HttpResponse` constant. This is just an ordinary class property, so you can ask a constant like `OK` for its code.

Functions

An enum class can implement functions, including overriding base ones like `toString()`:

```
enum class HttpResponse(val code: Int, val message: String) {
    OK(200, "OK"),
    MOVED_PERMANENTLY(301, "Moved Permanently"),
    NOT_MODIFIED(304, "Not Modified"),
    UNAUTHORIZED(401, "Unauthorized"),
    NOT_FOUND(404, "Not Found"),
    INTERNAL_SERVER_ERROR(501, "WTF?");
}
```

ENUMS AND SEALED CLASSES

```
    override fun toString() = message
}

fun main() {
    println(HttpResponse.INTERNAL_SERVER_ERROR.toString())
}
```

(from "[enum and Functions](#)" in the *Klassbook*)

The list of constants must be the first thing in the enum declaration — if we tried to have our `toString()` before the `OK`, we would fail with a compile error.

Note that in this case the comma-delimited list of constants is ended with a semicolon. If the only thing in the enum declaration is the list of constants, the overall closing brace of the enum is sufficient to end the list of constants. If you have other things after the constants, though, such as our `toString()` function, you need the semicolon to officially end the list of constants.

An enum class is intrinsically abstract, so you can define abstract functions as well, implementing them on individual constants:

```
enum class WizardStep {
    INTRO {
        override fun nextStep() = REGISTER
    },
    REGISTER {
        override fun nextStep() = PERMISSIONS
    },
    PERMISSIONS {
        override fun nextStep() = THANKS
    },
    THANKS {
        override fun nextStep() = THANKS
    };

    abstract fun nextStep(): WizardStep
}
```

Here, each `WizardStep` knows the next `WizardStep` in the sequence, by overriding an abstract `nextStep()` function declared for `WizardStep`. This gets a bit odd with the last step, as by definition the last step is last, so there is no “next step”. You might be tempted to return something like `null` from `nextStep()` on `THANKS...` we will see how that works in [an upcoming chapter](#).

Limitations

An enum class is allowed to implement interfaces, but it is not allowed to extend another class.

Also, akin to data classes, an enum cannot be open for extension... by classes outside of the enum class. Each of the enumerated constants (e.g., REGISTER, THANKS in the above example) is in effect a subclass of the enum class, complete with override methods to match the abstract ones in the enum class. But you cannot create new constants from outside the enum class, and you cannot create arbitrary other subclasses of the enum class.

Common Properties

Each enumerated constant has two properties, beyond those that you might declare yourself: `name` and `ordinal`. `name` returns the symbolic name that you gave the constant in your code, and `ordinal` returns the o-based index indicating where this constant appears in the list of constants. You can access `name` and `ordinal` as you can any other property:

```
enum class HttpResponse(val code: Int, val message: String) {
    OK(200, "OK"),
    MOVED_PERMANENTLY(301, "Moved Permanently"),
    NOT_MODIFIED(304, "Not Modified"),
    UNAUTHORIZED(401, "Unauthorized"),
    NOT_FOUND(404, "Not Found"),
    INTERNAL_SERVER_ERROR(501, "WTF?");

    override fun toString() = message
}

fun main() {
    println(HttpResponse.INTERNAL_SERVER_ERROR.toString())
    println(HttpResponse.INTERNAL_SERVER_ERROR.code)
    println(HttpResponse.INTERNAL_SERVER_ERROR.name)
    println(HttpResponse.INTERNAL_SERVER_ERROR.ordinal)
}
```

(from "[enum name and ordinal](#)" in the *Klassbook*)

This yields:

```
WTF?  
501  
INTERNAL_SERVER_ERROR  
5
```

The name could be useful in `toString()` implementations and similar scenarios. The ordinal is less likely to be useful — in particular, since it is position-dependent, it may be a bit fragile, as somebody reordering lines of your code might change the ordinal values for those constants.

Conversion

Each enum class is given a few functions to be called on the class itself. One is `valueOf()`. This returns a constant given the symbolic name that you gave the constant in your code. In other words, it works like the inverse of the `name` property — `name` gives you the symbolic name for a constant, while `valueOf()` gives you the constant for a symbolic name:

```
enum class HttpResponse(val code: Int, val message: String) {  
    OK(200, "OK"),  
    MOVED_PERMANENTLY(301, "Moved Permanently"),  
    NOT_MODIFIED(304, "Not Modified"),  
    UNAUTHORIZED(401, "Unauthorized"),  
    NOT_FOUND(404, "Not Found"),  
    INTERNAL_SERVER_ERROR(501, "WTF?");  
  
    override fun toString() = message  
}  
  
fun main() {  
    println(HttpResponse.valueOf("NOT_MODIFIED"))  
}
```

(from ["enum valueOf\(\)" in the Klassbook](#))

This results in:

```
Not Modified
```

`valueOf()` returns the `NOT_MODIFIED` constant. `println()` then calls `toString()` implicitly on that constant, and our overridden `toString()` function returns "Not Modified".

Iteration

Also, you can call `values()` on the enum class itself (e.g., `HttpResponse.values()`). This will allow you to iterate over all of the constant members of the enum class, in the order in which they were declared. Hence, this code:

```
enum class HttpResponse(val code: Int, val message: String) {
    OK(200, "OK"),
    MOVED_PERMANENTLY(301, "Moved Permanently"),
    NOT_MODIFIED(304, "Not Modified"),
    UNAUTHORIZED(401, "Unauthorized"),
    NOT_FOUND(404, "Not Found"),
    INTERNAL_SERVER_ERROR(501, "WTF?");

    override fun toString() = message
}

fun main() {
    for (constant in HttpResponse.values()) {
        println(constant)
    }
}
```

(from ["enum values\(\)" in the Klassbook](#))

results in:

```
OK
Moved Permanently
Not Modified
Unauthorized
Not Found
WTF?
```

Exhaustive when

We saw that you can use `when` as [an expression](#), where each one of the branches inside the `when` supply the value for the expression when that branch is true. However, one limitation that we saw was that you needed an `else` condition when using `when` as an expression, as for any possible condition, the `when` needs to generate a value.

One exception to that rule is an “exhaustive when” based on an enum class. If you have conditions for each enumerated constant, you do not need an `else` condition, since

by definition every possibility will have been handled by one of the other conditions.

For example, this script yields the same result as the one shown above:

```
enum class HttpResponse(val code: Int) {
    OK(200),
    MOVED_PERMANENTLY(301),
    NOT_MODIFIED(304),
    UNAUTHORIZED(401),
    NOT_FOUND(404),
    INTERNAL_SERVER_ERROR(501);

    override fun toString() = when(this) {
        OK -> "OK"
        MOVED_PERMANENTLY -> "Moved Permanently"
        NOT_MODIFIED -> "Not Modified"
        UNAUTHORIZED -> "Unauthorized"
        NOT_FOUND -> "Not Found"
        INTERNAL_SERVER_ERROR -> "WTF?"
    }
}

fun main() {
    for (constant in HttpResponse.values()) {
        println(constant)
    }
}
```

(from ["enum and when\(\)" in the Klassbook](#))

This time, `toString()` no longer just reads some property. We no longer have those properties, and instead use a `when` expression to get the human-readable message to go along with the HTTP response code. We do not need an `else` in the `when`, since every possible enumerated constant has its own condition. We have “exhausted” all possibilities with the specific conditions, and so this is an “exhaustive” `when`.

This particular example is really silly — having these messages as properties would be a better choice. However, you may have other places in your code where you need to branch based on an enum value, and so long as all possible values are accounted for, you do not need an `else`.

Sealed Classes

Sealed classes in Kotlin have nothing to do with [wax](#), [aquatic mammals](#), or

[musicians](#).

Instead, sealed classes are a way of limiting a class hierarchy. A class can only directly extend a sealed class if:

- In a regular project, it is in the same Kotlin file as is the sealed class itself
- In a REPL, it is a nested class inside of the sealed class

As a result, once that one file is read in by a Kotlin compiler, it immediately knows the complete possible set of all direct subclasses of the sealed class. This is a bit reminiscent of an enum class, which also has rules for what can subclass it (specifically, only its constant elements).

Um, Why?

A key limitation of an enum class is that each of its constant members is not only a subclass of the enum class, but is a singleton instance of that subclass. Going back to the `HttpResponse` scenarios from above, there is exactly one `OK` instance for the entire program. So an enum class limits not only where subclasses can reside but where instances can be declared.

In contrast, a sealed class controls the class hierarchy, but it does not limit instance creation. There can be N instances of a subclass of a sealed class, each with its own data.

However, since the subclasses of the sealed class are readily identifiable — they are all in the same Kotlin source file — we get some of the same benefits that we get with enum classes, notably the exhaustive when support.

Basic Declaration and Usage

To create a sealed class, start by adding the `sealed` keyword:

```
sealed class BrowserLocation(val url: String)
```

Sealed classes are abstract, so you can add anything that you want that works with an abstract class: properties, concrete functions, abstract functions, etc.

Then, add nested classes and objects that extend from the sealed class:

ENUMS AND SEALED CLASSES

```
sealed class BrowserLocation(open val url: String) {
    object HomePage : BrowserLocation("https://commonsware.com")

    data class Bookmark(override val url: String, val name: String) : BrowserLocation(url)

    data class HistoryEntry(override val url: String, val title: String, val lastVisited: String) :
BrowserLocation(url)
}

fun main() {
    println(BrowserLocation.HomePage.url)

    val bookmark = BrowserLocation.Bookmark("https://kotlinlang.org", "Kotlin!")

    println(bookmark)
}
```

(from ["Sealed Classes" in the Klassbook](#))

You then refer to them the same as you would any other nested classes or objects (e.g., `Browser.HomePage`). So, running this script results in:

```
https://commonsware.com
Bookmark(url=https://kotlinlang.org, name=Kotlin!)
```

However, there is nothing particularly magical about `BrowserLocation` being sealed so far. You could replace `sealed` with `abstract` and get the same results.

Exhaustive When

A sealed class, like an enum, supports an exhaustive when. All possible classes and objects that directly extend the sealed class are known when the sealed class is compiled. So long as your when covers all of those possibilities, you do not need an else clause:

```
sealed class BrowserLocation(open val url: String) {
    object HomePage : BrowserLocation("https://commonsware.com")

    data class Bookmark(override val url: String, val name: String) : BrowserLocation(url)

    data class HistoryEntry(override val url: String, val title: String, val lastVisited: String) :
BrowserLocation(url)
}

fun main() {
    val location: BrowserLocation = BrowserLocation.Bookmark("https://kotlinlang.org", "Kotlin!")

    val title = when (location) {
        BrowserLocation.HomePage -> "Home"
        is BrowserLocation.Bookmark -> location.name
        is BrowserLocation.HistoryEntry -> location.title
    }
}
```

```
println(title)
}
```

(from "[Sealed Classes and when](#)" in the *Klassbook*)

This prints:

```
Kotlin!
```

Smart Casts

That too may seem somewhat unremarkable. However, something very interesting is going on with the latter two branches of the when.

location is a `BrowserLocation` variable, because we explicitly set that to be the data type:

```
val location: BrowserLocation = BrowserLocation.Bookmark("https://kotlinlang.org",
"Kotlin!")
```

The first branch of the when compares location with `BrowserLocation.HomePage` — if location is that singleton object, then we return "Home" as the title. That is fairly normal.

However, the second branch checks to see if the type of location is `BrowserLocation.Bookmark`. If it is, then we return the name property. But location is a `BrowserLocation` variable, and `BrowserLocation` does not have a name property. If we try referencing it directly, we get a compile error. So, this:

```
println(location.name)
```

...results in:

```
error: unresolved reference: name
println(location.name)
```

So, why does it work in the when?

That is because Kotlin realizes that if we are executing `location.name` in that branch, location must be a `BrowserLocation.Bookmark` — otherwise, we would have failed the `is` check and would not be taking that branch. Since Kotlin knows that location must be a `BrowserLocation.Bookmark`, Kotlin knows that it is safe to reference the name property.

The same approach is used for the third branch. We can safely reference `location.title` there, because if we are executing that branch, we know that `location` is a `BrowserLocation.HistoryEntry`, which has a `title` property.

This is another variation on Kotlin’s “smart casts” that we saw [earlier in the book](#). The Kotlin compiler can use knowledge of how code gets executed to allow you to avoid manual casts, because what you want to do happens to be valid for objects in that state.

Smart casts are *very* useful, particularly when dealing with `null` values, as we will explore [in an upcoming chapter](#).

Scenario: Valid and Invalid Data

A common pattern involving sealed classes comes when interacting with some server or other external source of data. Usually, there are two main outcomes from such an interaction:

- We get valid data
- We receive some sort of error, either from the source itself (e.g., a Web service responding with a JSON object indicating that our request failed) or from infrastructure around that source (e.g., an `IOException` from being unable to reach the Internet)

One way to model those responses is to have a sealed class hierarchy that represents both types of outcomes. If we are going to treat all errors the same, we could use the approach shown with `BrowserLocation`, where we have a mix of classes and objects in the sealed class:

```
sealed class ThingyResponse {
    data class Thingy(val something: String, val somethingElse: Int) : ThingyResponse()

    data class OtherThingyThatWeMightGet(
        val like: Float,
        val whatever: String,
        val dude: Boolean
    ) : ThingyResponse()

    object Invalid : ThingyResponse()
}
```

ENUMS AND SEALED CLASSES

```
class WebServiceApi {
    fun requestThingy(): ThingyResponse = ThingyResponse.Invalid
}
```

Here, our `WebServiceApi` can return a `ThingyResponse` from `requestThingy()`, and that can encompass both positive and negative outcomes. Right now, that function is stubbed out by always returning `ThingyResponse.Invalid`, but we could have a full Web service request here that parses results, generates `ThingyResponse.Thingy` or `ThingyResponse.OtherThingyThatWeMightGet` objects, and so on.

Callers of `requestThingy()` and anything else that gets a `ThingyResponse` can use an exhaustive when to handle all of the possibilities.

Another variant is to have `Invalid` be a class instead of a singleton object. That class can then hold details of what went wrong:

- Error code from the server
- An [exception](#) that was raised by the network I/O
- Etc.

Scenario: Loading/Content/Error

A common pattern for a UI is:

- We need to show some loading state, such as a progress spinner, while disk or network I/O is ongoing
- When we get the content to display, display that
- If something went wrong, show some sort of error state

Similar to the valid-and-invalid data scenario above, a sealed class lets us model these three states with a single core type:

```
sealed class SomethingViewState {
    object Loading : SomethingViewState()

    data class Content(val goodStuff: List<Stuff>) : SomethingViewState()

    object Error : SomethingViewState()
}
```

For example, in Android app development, you might emit instances of this `SomethingViewState` from a `ViewModel`, perhaps using the Jetpack `LiveData` class or

StateFlow from Kotlin's coroutines. Your UI layer (activity, fragment, composable) could observe those states and use an exhaustive when to handle all possible state sub-types (Loading, Content, Error).

Limitation: Location

Direct subclasses of the sealed class must be in the same file that contains the sealed class itself. They do not necessarily need to be nested inside the sealed class, though.

So, normally, this is perfectly valid... in most Kotlin environments:

```
sealed class BrowserLocation(open val url: String)

object HomePage : BrowserLocation("https://commonsware.com")

data class Bookmark(override val url: String, val name: String) : BrowserLocation(url)

data class HistoryEntry(override val url: String, val title: String, val lastVisited: String) :
    BrowserLocation(url)

fun main() {
    val location: BrowserLocation = Bookmark("https://kotlinlang.org", "Kotlin!")

    val title = when (location) {
        HomePage -> "Home"
        is Bookmark -> location.name
        is HistoryEntry -> location.title
    }

    println(title)
}
```

(from "[Non-Nested Sealed Classes](#)" in the *Klassbook*)

Early versions of Kotlin required sealed class sub-types to be nested in the sealed class. Also, some REPL environment require nesting.

However, most Kotlin projects are not written in a REPL and are using modern versions of Kotlin, so having a subclass of the sealed class be a peer of the sealed class is fine. The “must-be-nested” rule applies for `.kts` files, not `.kt` files.

If you declare a direct subclass of the sealed class to be open, though, you can extend that class normally, even from other source files. So, in a regular Kotlin project (not a REPL), if you have one source file with:

```
import java.time.Instant

sealed class BrowserLocation(open val url: String)
```

ENUMS AND SEALED CLASSES

```
object HomePage : BrowserLocation("https://commonsware.com")

open class Bookmark(override val url: String, open val name: String) :
    BrowserLocation(url)

data class HistoryEntry(override val url: String, val title: String, val lastVisited:
    Instant) : BrowserLocation(url)
```

...and in another source file, you have:

```
class BrokenSeal(override val url: String, override val name: String) : Bookmark(url,
    name)
```

...then you are OK. Here, `Bookmark` is marked as an open class, with both of its constructor properties being open as well (`url` is declared as open up in `BrowserLocation`). As a result, we can extend `Bookmark` from a separate Kotlin source file, as `BrokenSeal` does.

Generics

If you have done Java development, you are no doubt familiar with generics, otherwise known as “death by a thousand angle brackets”:

```
class TouchedByAnAngle<T> {
    private T thingy;

    public T getThingy() {
        return thingy;
    }

    public void setThingy(T replacement) {
        thingy = replacement;
    }
}
```

If you like Java’s generics, rest assured that Kotlin has a very similar system.

If you keep getting confused by Java’s generics, rest assured that, in time, you will be just as confused by Kotlin’s implementation.

And, if you are not that familiar with Java’s generics... that’s what the next section is for.

OK, What Are These For Again?

Kotlin, like Java, is “strongly typed”. This means each variable, property, parameter, and return value have a specific type, such as `ArrayList` or `Axolotl` or `Thingy`. The compiler will attempt to ensure that you only provide objects of valid types for these things.

We saw that back in [the chapter on classes](#):

```
open class Animal
class Frog : Animal()
class Axolotl : Animal()
fun main() {
    val critter: Animal = Frog()
    if (critter is Frog) println("Ribbit!") else println("Ummm... whatever noise an axolotl makes!")
}
```

(from "[Checking Inheritance Via is](#)" in the *Klassbook*)

Here, `critter` is a variable of type `Animal`. We can assign an instance of a `Frog` to `critter`, because `Frog` extends `Animal`, and so `Frog` is a compatible type. But we could not assign an `Int` to `critter`, as Kotlin's `Int` type does not extend `Animal`.

Now, suppose that we wanted a list of animals.

We could use the `listOf()` global function, supplied by Kotlin's standard library, to create a list of animals:

```
val critters = listOf(Frog(), Axolotl())
```

From a type safety standpoint, though, it really is a `List` of `Animal` objects:

```
val critters: List<Animal> = listOf(Frog(), Axolotl())
```

Here we are saying that `critters` can only hold `Animal` objects. We cannot put an `Int` into the list, as `Int` is not an `Animal`.

Types like the `List` interface and the `ArrayList` implementation of `List` can use generics to constrain the type of objects that they work with. Therefore, when we are trying to work with a list of animals, we do not have to worry about accidentally encountering an integer, a string, or something else.

Instantiation with Generics

We have a few options for creating objects and declaring generic types.

The Formal Way

The official way for all of this is to declare the type both where we are using it (e.g., a property) and when creating the instance:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val critters: ArrayList<Animal> = ArrayList<Animal>()

    critters.add(Frog())
    critters.add(Axolotl())
}
```

Here, we are very specifically stating that the `critters` property holds an `ArrayList` of `Animal` and is being initialized with an `ArrayList` of `Animal`.

If we wanted, we could skip the generic type on the initializer:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val critters: ArrayList<Animal> = ArrayList()

    critters.add(Frog())
    critters.add(Axolotl())

    println(critters::class)
    println(critters.map { it::class.toString() }.joinToString())
}
```

(from ["Generics" in the Klassbook](#))

Here we have two additional lines, so `Klassbook` has some output:

- We print the class of `critters`. This will show up as `ArrayList`, not `ArrayList<Animal>`, because generics are only used at compile time.

GENERICIS

- We print the classes of the members of the list, which will show up as `class Frog`, `class Axolotl`

We could try to substitute a supertype, such as `List`, for the property type:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val critters : List<Animal> = ArrayList<Animal>()

    critters.add(Frog())
    critters.add(Axolotl())
}
```

...except that this does not work:

```
error: unresolved reference: add
critters.add(Frog())
           ^
error: unresolved reference: add
critters.add(Axolotl())
           ^
```

As it turns out, Kotlin's `List` type is immutable: you cannot add new objects to the list after initially creating it. We will explore immutable types more [in an upcoming chapter](#).

Fortunately, Kotlin also has a `MutableList` interface that `ArrayList` happens to implement, so we can use:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val critters : MutableList<Animal> = ArrayList<Animal>()
}
```

```
critters.add(Frog())
critters.add(Axolotl())
}
```

Implied Types

Our original example, using `listOf()`, did not declare a type. Nor does this slightly-modified example:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val critters = mutableListOf(Frog(), Axolotl())

    critters.add("ribbit!")
}
```

The two changes are:

- We use `mutableListOf()`, which is a global function from Kotlin's standard library that creates a `MutableList` with some initial contents; and
- We try adding a string to that list

This fails with a compile error:

```
error: type mismatch: inferred type is String but Animal was expected
critters.add("ribbit!")
              ^
```

What happened is that the Kotlin compiler looked at the type of objects that we passed into `mutableListOf()` and determined the common supertype, then decided that `critters` must be a variable of that type. In this case, `Frog()` and `Axolotl()` are both `Animal` objects, so Kotlin will treat `critters` as if it were a `MutableList` of `Animal`.

The key is that Kotlin will figure out the common supertype. If we really do want to allow strings in our list, we could teach Kotlin that by adding a string at the outset:

GENERICIS

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val critters = mutableListOf(Frog(), Axolotl(), "ribbit!")

    critters.add("[insert axolotl sound here]")

    println(critters::class)
    println(critters.map { it::class.toString() }.joinToString())
}
```

(from ["Implied Generic Type" in the Klassbook](#))

This compiles just fine, because Kotlin has decided that `critters` is a variable of some type that `Frog`, `Animal`, and `String` all have in common. As it turns out, that Kotlin type is called `Any`, and it serves as the base class for all Kotlin classes (akin to how `Object` is the base class for all Java classes).

This is not limited to some sort of magic global function:

```
import java.util.concurrent.atomic.AtomicReference

data class Physicist(val firstName: String)

val oppenheimer = AtomicReference(Physicist("Robert"))
```

Here, we use Java's `AtomicReference` class, which provides a thread-safe way to access a value. We do not explicitly state that `oppenheimer` is an `AtomicReference` of `Physicist` — Kotlin figures that out. If we tried calling `oppenheimer.set("Albert")`, it would fail with:

```
error: type mismatch: inferred type is String but Test.Physicist! was expected
oppenheimer.set("Albert")
                ^
```

So, in general, you can skip the type declaration of a variable, so long as the type that Kotlin decides to use, based on your initializer, is the type that you want to use. Otherwise, declare the type explicitly yourself.

Applying Generics to Classes and Interfaces

You may want to support generics in your own classes and interfaces. This works a lot like how it does in Java, where you use T notation to indicate where you accept a varying type.

Use in Class Declaration

For example, you can create a class that can work with a particular type:

```
open class Animal
class Frog : Animal()
class Axolotl : Animal()
data class Transport<T>(var passenger: T)
fun main() {
    val kermit = Frog()
    val critterCarrier = Transport(kermit)
    println(critterCarrier.passenger::class)
}
```

(from "[Your Own Generic Classes](#)" in the *Klassbook*)

Transport can handle an object of any type, but a particular instance of Transport can only handle objects of a single type (or subtypes, where applicable). When we pass a Frog instance into the Transport constructor, we lock the resulting Transport to transporting Frog objects. If we attempt to reassign passenger with another type:

```
critterCarrier.passenger = "This is not an Animal"
```

...we get a compile error:

```
error: type mismatch: inferred type is String but Test.Frog was expected
critterCarrier.passenger = "This is not an Animal"
                             ^
```

What the Transport can support will be based on the type of the variable or property handed to it. So, we can give a Transport a bit more flexibility by passing in an Animal, instead of a Frog, to it:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

data class Transport<T>(var passenger: T)

fun main() {
    val kermit: Animal = Frog()
    val critterCarrier = Transport(kermit)

    println(critterCarrier.passenger::class)

    critterCarrier.passenger = Axolotl()

    println(critterCarrier.passenger::class)
}
```

(from "[As Generic As You Want](#)" in the *Klassbook*)

This works because now kermit is typed as an `Animal`, so `critterCarrier` is a `Transport` of `Animal`, so we can replace our `Frog` with an `Axolotl` if desired.

Use with Supertypes

We could also create a subtype that declares support for a specific type:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

interface Transport<T> {
    abstract fun getPassenger(): T
}

data class Van(val animal: Animal) : Transport<Animal> {
    override fun getPassenger() = animal
}

fun main() {
    val kermit = Frog()
    val critterCarrier = Van(kermit)
}
```

```
println(critterCarrier.getPassenger()::class)
}
```

(from "[Generics and Supertypes](#)" in the *Klassbook*)

Here, while `Transport` can handle any type, `Van` is an implementation of `Transport` that is restricted to transporting `Animal` objects.

Note that we can use the `T` type placeholder for properties, parameters, and — in the case of `getPassenger()` — return values.

Upper Bounds

A simple declaration of `T` could be any type. Sometimes, we need to limit it to only be certain types. The simplest way to do that is to declare `T` a bit like a class or interface, using a colon and a type to declare what `T` must inherit from:

```
open class Thingy

open class Animal : Thingy()

class Frog : Animal()

class Axolotl : Animal()

interface Transport<T : Thingy> {
    abstract fun getPassenger(): T
}

data class Van(val animal: Animal) : Transport<Animal> {
    override fun getPassenger() = animal
}

fun main() {
    val kermit = Frog()
    val critterCarrier = Van(kermit)

    println(critterCarrier.getPassenger()::class)
}
```

(from "[Generics and Upper Bounds](#)" in the *Klassbook*)

Now `Transport` cannot transport anything — it is limited to instances of `Thingy`. Since an `Animal` is a `Thingy`, we can still transport animals. But we cannot create a

GENERICIS

Transport of String, because a String does not extend from Thingy. If we try anyway:

```
open class Thingy

interface Transport<T : Thingy> {
    abstract fun getPassenger(): T
}

data class StringVehicle(val value: String) : Transport<String> {
    override fun getPassenger() = value
}
```

...we wind up with a compile error:

```
error: type argument is not within its bounds: should be subtype of 'Test.Thingy'
data class StringVehicle(val value: String) : Transport<String> {
                                     ^
```

Generics, WTF?

Generics are complicated, and that ties into some esoteric-looking Kotlin syntax that we will explore much later in the book:

- When you see the out keyword, such as in class Something<out T>, that is Kotlin's approach for [declaring "covariance"](#), stipulating that an instance of Something is allowed to *produce* T objects (e.g., return them from functions) but not *consume* them (e.g., accept them as parameters)
- When you see the in keyword, such as in class Something<in T>, that is Kotlin's approach for [declaring "contravariance"](#), stipulating that an instance of Something is allowed to *consume* T objects but not *produce* them
- When you see <reified T>... well, [that is difficult to explain](#)

Exceptions

In an ideal world, nothing would ever go wrong with our apps.

In an ideal world, the author of this book would have a full head of hair.

This is not an ideal world.

As with most modern programming languages, when things go wrong, exceptions get raised. We will need to be able to handle those exceptions when they occur, one way or another.

Kotlin's exception system is very similar to that of Java, except that there are no [checked exceptions](#).

Catching Exceptions

A classic scenario where our apps can fail is with a `NullPointerException`.

We will explore nullability in greater detail [in an upcoming chapter](#), as Kotlin takes steps to try to minimize a `NullPointerException`. However, you can still wind up crashing if you try accessing null inappropriately, such as we are doing here:

```
var thisIsReallyNull: String? = null

println(thisIsReallyNull)
println(thisIsReallyNull!!.length)
```

(from "[Exceptions](#)" in the *Klassbook*)

This compiles fine, but it crashes at runtime with a `NullPointerException`. We will

see exactly what ? and !! do [in the chapter on nullability](#).

For now, though, let's focus on the fact that we are getting an exception. In this case, the exception is very artificial, but there are many places where an exception may be expected somewhat. For example, if you try accessing the Internet, there may be all sorts of problems resulting in all sorts of exceptions.

Traditional Try-Catch

Kotlin supports try and catch, using syntax very similar to that of Java:

```
try {
    var thisIsReallyNull: String? = null

    println(thisIsReallyNull)
    println(thisIsReallyNull!!.length)
}
catch (e: Exception) {
    println("ERROR: '$e'")
}
```

(from "[try and catch](#)" in the *Klassbook*)

To catch an exception, you:

- Wrap the code that might trigger the exception in a block, preceded by try, then
- Follow that block with a catch block, where you indicate what exception you are expecting

Here, we wrap the null reference in the try block. If — or, rather, when — that code throws an exception, we will get control in our catch block, where we can do something.

Between the catch keyword and the block is a parameter-style declaration in parentheses, indicating:

- What sort of exception we are looking to catch
- What name to give this exception, that we can use inside the code in the catch block to examine the exception itself

Here, anything that extends from Exception would be caught and handled by our catch block.

Cascading Catches

If you want to handle different types of exceptions differently, you can have multiple catch blocks:

```
try {
    var thisIsReallyNull: String? = null

    println(thisIsReallyNull)
    println(thisIsReallyNull!!.length)
}
catch (npe: NullPointerException) {
    println("You tried doing something unfortunate with null. Stop that.")
}
catch (e: Exception) {
    println("ERROR: '$e'")
}
```

(from "[Multiple catch Blocks](#)" in the *Klassbook*)

Here, when the null reference throws an `NullPointerException`, we will execute the first catch block. Otherwise, if it were to throw some other type of `Exception`, we will execute the second catch block.

Note, though, that Kotlin does not support multiple types in a single catch, the way you can in Java:

```
try {
    // something that might throw multiple types of exceptions
}
catch (ThisException | ThatException ex) {
    // handle those two exception types
}
```

What You Might Catch

The details of what exceptions can be thrown will vary by circumstance. In particular, the Kotlin environment will play a major role here. For example, `java.lang.ArithmeticException` is a Java thing that you might catch on Kotlin/JVM. Other Kotlin environments, such as Kotlin/JS, will not know what `java.lang.ArithmeticException` is.

Even within an environment, the details may vary. For example, in Java, `Exception` is a subclass of `Throwable`. An `Exception` is something that the Java language designers

felt that apps should be able to handle. An `Error`, though, is another type of a `Throwable` that the language designers considered to be unrecoverable. For example, an `OutOfMemoryError` is something that the Java designers thought was something that should result in the death of your process.

Typically, in a catch block, you identify exceptions that you want to catch because you have specific local ways to recover from it. If you need a true “catch everything” catch block, in Kotlin/JVM, you would catch a `Throwable`, not an `Exception`.

A Missed Catch

An exception that is not caught cascades up the “call stack” to whatever the next function is:

```
fun itsGonnaBlow() {
    var thisIsReallyNull: String? = null

    println(thisIsReallyNull)
    println(thisIsReallyNull!!.length)
}

fun main() {
    try {
        itsGonnaBlow()
    }
    catch (npe: NullPointerException) {
        println("You tried doing something unfortunate with null. Stop that.")
    }
    catch (e: Exception) {
        println("ERROR: '$e'")
    }
}
```

(from ["Exceptions and the Call Stack" in the Klassbook](#))

Here, `itsGonnaBlow()` triggers the exception, but we do not have a `try/catch` structure in that function. So, when `itsGonnaBlow()` throws the exception, it gets passed to whoever called `itsGonnaBlow()`, and so on.

Try as an Expression

Like `if` and `when`, `try` (with its associated catch blocks) represents an expression, and you can use its result as you would any other expression, such as assigning it to a variable or property.

EXCEPTIONS

In the case of `if` and `when`, the result of the expression is based on which of the branches you take. In the case of `try`, the result of the expression is:

- The value of the last expression of the `try` block, if there is no exception raised in that block, or
- The value of the last expression of the `catch` block, if one of the `catch` blocks caught an exception

```
fun lengthifier(nullableButNotReally: String?): Int {
    return nullableButNotReally!!.length
}

fun main() {
    val result = try {
        lengthifier(null)
    }
    catch (e: Exception) {
        -1
    }

    println(result)
}
```

(from ["try as an Expression" in the Klassbook](#))

Here, we compute the length of a `String` in a function, and we wrap that function call in a `try/catch` structure. `result` will either be:

- The length of the string, or
- `-1` if we crash, such as with a `NullPointerException`

In this case, we are trying to get the length of `null`, so we get `-1` as the result of our `try` expression.

Finally, Don't Forget `finally`

As with Java, Kotlin also offers a `finally` block that you can chain onto the end of your `try/catch` structure. This block's code will be executed either:

- After the `try` block's contents, if there are no exceptions thrown by that block, or
- After the `catch` block's contents, if that `catch` caught an exception raised by the `try` block

EXCEPTIONS

Typically, you use the `finally` block to ensure that everything is cleaned up from whatever may (or may not) have happened in the `try`:

```
try {
    // TODO something that might blow up
}
catch (e: Exception) {
    // TODO something to deal with the blow up when it blows up
}
finally {
    // TODO clean up afterwards
}
```

Note that if you use the `try` as an expression, the `finally` block has no impact on the result of the expression. The result will still be based on the `try` result or the `catch` result — the `finally` code is executed but does not change the result.

Raising Exceptions

Frequently, we do not need to raise our own exceptions. We just deal with the ones that arise from the code that we write, where those exceptions get raised by Kotlin, libraries, etc.

However, from time to time, you might want to throw an exception yourself.

Here, once again, Kotlin works like Java. To throw an exception:

- Create an instance of an exception class, such as `IllegalArgumentException`
- Use that with the `throw` keyword

```
fun lengthifier(nullAllowedButNotReally: String?): Int {
    if (nullAllowedButNotReally == null) throw Exception("Please do not pass null to me!")

    return nullAllowedButNotReally!!.length
}

fun main() {
    println(lengthifier(null))
}
```

(from "[Throwing Exceptions](#)" in the *Klassbook*)

Here, we manually check to see if we received `null` in `lengthifier()`, and we throw our own `Exception` if that is the case. So, if we run this, we get our `Exception`:

```
Exception: Please do not pass null to me!
```

Defining Exceptions

You might have a need to create your own custom exceptions.

For example, suppose you are calling a Web service. You will get errors back from that Web service in a variety of ways, such as:

- Error HTTP result codes
- Error information encoded in the JSON (or whatever) payload sent back by the server

You might decide to convert those error conditions into exceptions.

As with Java, you can create subclasses of `Exception` to represent your custom exceptions:

```
class GoAway(message: String) : Exception(message)

fun lengthifier(nullAllowedButNotReally: String?): Int {
    if (nullAllowedButNotReally == null) throw GoAway("Please do not pass null to me!")

    return nullAllowedButNotReally!!.length
}

fun main() {
    val result = try {
        lengthifier(null)
    }
    catch (away: GoAway) {
        -2
    }
    catch (e: Exception) {
        -1
    }

    println(result)
}
```

(from "[Custom Exception Classes](#)" in the *Klassbook*)

Here, we create a custom `GoAway` exception and throw it rather than an `Exception`.

If you have a set of related exception types, a class hierarchy (using open, abstract, or sealed) may be useful:

```
open class MathBoom(message: String) : Exception(message)

class DivideByZeroBoom : MathBoom("Please do not divide by zero")
```

EXCEPTIONS

```
fun divider(numerator: Double, denominator: Double): Double {
    if (denominator == 0.0) throw DivideByZeroBoom()

    return numerator / denominator
}

val result = try {
    divider(1.0, 0.0).toString()
}
catch (e: MathBoom) {
    "whatever"
}

println(result)
```

Checked vs. Unchecked Exceptions

While the concept of checked exceptions is not unique to Java, it is a feature most commonly associated with that language. In Java, a checked exception is declared as part of a method, which indicates what sorts of exceptions it might throw:

```
public void doSomethingPotentiallyTroublesome(File f) throws FileNotFoundException {
    if (!f.exists()) throw new FileNotFoundException(f.getAbsolutePath()+" does not exist!");

    // TODO other stuff, now that we have a valid value
}
```

Callers of a method that throws checked exceptions *must* handle those exceptions somehow, either by exposing those exceptions as part of their own method or by catching them and doing something, such as throwing some type of a `RuntimeException`, as those are unchecked exceptions:

```
public void troublesomeWrapper(File f) {
    try {
        doSomethingPotentiallyTroublesome(f);
    }
    catch (FileNotFoundException e) {
        throw new IllegalStateException("I'm in a bad state", e);
    }
}
```

Checked exceptions have been debated a lot over the years. It appears that the consensus opinion is that checked exceptions were an interesting idea that caused

EXCEPTIONS

more problems than they were worth.

As a result, Kotlin does not offer checked exceptions. Instead, Kotlin treats all exceptions as unchecked, which is the approach taken by many other programming languages.

Annotations

Most Java developers are familiar with annotations. These are the @-prefixed values that decorate classes, methods, fields, and so on:

```
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    @Test
    public void useAppContext() {
        Context appContext=InstrumentationRegistry.getTargetContext();

        assertEquals("com.commonware.jetpack.hello", appContext.getPackageName());
    }
}
```

Here, @RunWith and @Test are annotations. These happen to be from JUnit (specifically JUnit 4) and are used to teach a JUnit test runner:

- How to run a particular test class (“this test class needs to be run with the assistance of AndroidJUnit4”)
- Which methods represent test methods that should be executed when running the test

Kotlin’s annotations are very similar to Java’s, with a few syntax differences.

Where Annotations Come From

The Java example above shows a pair of annotations from JUnit 4. JUnit 4 is annotation-based; it supplies quite a few annotations.

Overall, there are a few different places where annotations can come from.

Kotlin

Kotlin has some annotations that are part of the Kotlin standard library. One that we will examine in this chapter is Kotlin's take on the `@Deprecated` annotation, used to denote some code (class, method, field, etc.) that should no longer be used.

Java

If you are using Kotlin/JVM (e.g., for Android development), Java's annotations are also available to you. You are unlikely to use many of them directly yourself — for example, Java's `@Override` annotation is replaced by Kotlin's `override` keyword. However, technically, they are available to you.

Libraries

Most annotations come from libraries. These libraries will come with “annotation processors” that know how to scan a code base for annotations and do something useful with them, either:

- At compile time for code generation, or
- At runtime to assist with code execution

JUnit 4 falls in the latter category, using annotations to help guide how to execute your tests. Dagger — a popular dependency injection framework for Java — uses them at compile time for code generation.

You!

You are welcome to create your own annotations.

Most developers will have little need to do this. Some will create annotations designed to be used by other annotation processors, such as using custom annotations with Dagger to help guide its dependency injection rules. And a few enterprising developers will create their own annotation processors. This book will not get into how to implement an annotation processor, though we will see some simple custom annotations.

Applying Annotations

First, though, let's explore how we can annotate our Kotlin code, using annotations

that somebody else created.

Basic Syntax

For simple use of annotations, the syntax is nearly identical to that of Java. For example, here is the Kotlin equivalent to the JUnit 4 example shown above:

```
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        val appContext = InstrumentationRegistry.getTargetContext()
        assertEquals("com.commonware.jetpack.hello", appContext.packageName)
    }
}
```



You can learn more about JUnit4 in Android in the "Touring the Tests" chapter of [Elements of Android Jetpack!](#)

As with Java, Kotlin annotations are prefixed with @ and are used to decorate classes, functions, properties, etc. Some annotations take parameters, such as the @RunWith annotation — we will examine this more closely [later in this chapter](#).

And, as with Java, whitespace does not matter. An annotation does not have to appear on a separate line from what it annotates. The above example could be rewritten as:

```
@RunWith(AndroidJUnit4::class) class ExampleInstrumentedTest {
    @Test fun useAppContext() {
        val appContext = InstrumentationRegistry.getTargetContext()
        assertEquals("com.commonware.jetpack.hello", appContext.packageName)
    }
}
```

Specialized Scenarios

Annotating classes, functions, and properties is fairly straightforward: just put the annotation immediately before the element that it is annotating.

However, there are a few things that can be annotated that may not be quite as

obvious.

Constructors

Most Kotlin classes have a single constructor, written with the constructor parameter list immediately after the class name:

```
class Foo(var count: Int) {
    fun something() {
        count += 1
        println("something() was called $count times")
    }
}
```

As we saw back in [the chapter on classes](#), this is a shorthand replacement for the formal approach:

```
class Foo constructor(var count: Int) {
    fun something() {
        count += 1
        println("something() was called $count times")
    }
}
```

If you need to annotate a constructor, you will need to use the formal approach, so you can place the annotation immediately before the constructor keyword:

```
class Foo @MyAnnotation constructor(var count: Int) {
    fun something() {
        count += 1
        println("something() was called $count times")
    }
}
```

Lambda Expressions

It is possible to annotate a lambda expression, by putting the annotation immediately before the opening brace:

```
val lambdaLambdaLambda = @MyAnnotation { doSomethingHere() }
```

Under the covers, a lambda expression “expands” into an instance of a class with a single `invoke()` function, with the body of the lambda expression forming the body

of that `invoke()` function. Annotating the lambda expression, in effect, annotates that `invoke()` function.

Annotation Parameters

Some annotations take parameters, akin to a class constructor. The syntax is similar to class constructors with some minor differences. However, most of the time, they will look akin to how annotations appear in Java. So, for example, the simplest form of using the `@Deprecated` annotation just passes in a `String` that is the explanation for the deprecation:

```
@Deprecated("do not use this, because reasons")
fun thisSeemsPerfectlyFine() {
    // do something
}
```

Named Parameters

You do not have to use named parameters with Kotlin annotations. This runs counter to Java, where there are scenarios where you have to name the parameters used in annotations.

However, if you want to use named parameters, you are welcome to do so:

```
@Deprecated(message = "do not use this, because reasons")
fun thisSeemsPerfectlyFine() {
    // do something
}
```

Arrays

Sometimes an annotation will take an array of values for a parameter, rather than a single value. For that, you have two options:

- Use the `arrayOf()` global function:

```
@Index(arrayOf("otherId", "yetAnotherId"))
```

- Use array literal syntax (square-bracket notation):

```
@Index(["otherId", "yetAnotherId"])
```


ANNOTATIONS

Array literal syntax was added to Kotlin in version 1.2, so some older materials might not mention it as an option.

Nested Annotations

It is possible that one of the parameters to an annotation is another annotation. In that case, the nested annotation does not get the @ prefix:

```
@Deprecated("do not use this, because reasons", ReplaceWith("thisIsMuchBetter()"))
fun thisSeemsPerfectlyFine() {
    // do something
}
```

This is different than Java's approach, which retains the @ prefix on nested annotations.

Class References

Sometimes, a parameter to an annotation is a class that is to be used by the annotation processor.

For example, in Android development, Room is a library for mapping Java classes to relational database tables, specifically for Android's SQLite database. Room has a RoomDatabase class that needs to be annotated with a @Database annotation. That annotation, in turn, takes an array of classes that represent the "entities" (the Java model classes that will have corresponding tables in the database).

For cases like this, we use simple ::class notation to identify the class:

```
@Database(entities = [SimpleEntity::class], version = 1)
abstract class SimpleDatabase : RoomDatabase() {
    // TODO add in the rest of the stuff that Room needs
}
```

Here, ::class identifies the Kotlin class (i.e., SimpleEntity::class is the Kotlin SimpleEntity class).

In reality, this annotation needs a *Java* class, but that is handled by the Kotlin compiler when it processes annotations. We will explore the difference between Kotlin class objects and Java class objects [later in the book](#).

Defining Annotations

Occasionally, you may have the need to create your own custom annotations.

In Kotlin, annotations are simply classes defined using the annotation keyword:

```
annotation class MyAnnotation
@MyAnnotation
class Foo
```

You are welcome to add constructor parameters to your annotation:

```
annotation class MyAnnotation(val isThisGood: Boolean)
@MyAnnotation(isThisGood = true)
class Foo
```

However, there are limits on the data types. For Kotlin/JVM development, Kotlin annotations get converted into Java annotations, and those have limits on their data types. You can use:

- Int and Long
- Float and Double
- Boolean
- String
- any enum that you define
- any other annotation
- classes (data type is KClass)

Kotlin Essentials

Nullability

The dreaded `NullPointerException` has been the bane of many Java developers. And Java actually handles `null` values relatively well, compared to languages like C/C++, where null pointers can have more damaging effects.

The designers of Kotlin elected to try to do something about it, by adding nullability to types, so that developers can be more aware of where `null` is (and is not) allowed, through compile-time type checking. It is not a perfect solution, but it does go a long way towards eliminating `NullPointerException` and related bugs.

Introducing Nullable Types

So far in this book, we have used types like `Int`, `Boolean`, `String`, and `Axolotl`. These types are normal Kotlin types... and they do not allow `null`.

So, if you try this:

```
val notGonnaHappen : String = null
```

...you get:

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

This is not merely a limitation of assignments. Any place where the type does not allow `null`, you cannot use a `null` value. So, this too fails with a compile error:

NULLABILITY

```
fun gotNoTimeForNull(parameter: String) {
    println(parameter)
}

fun main() {
    gotNoTimeForNull(null)
}
```

To say that `null` is a valid value, you append `?` to the type, giving you types like `Int?`, `Boolean?`, `String?`, and `Axolotl?`. So, this works:

```
val notGonnaHappen : String? = null

fun gotNoTimeForNull(parameter: String?) {
    println(parameter)
}

fun main() {
    gotNoTimeForNull(notGonnaHappen)
}
```

(from "[Nullable Types](#)" in the *Klassbook*)

You might be worried about that `println(parameter)` statement. If `parameter` can be `null`, might that crash with a `NullPointerException`? The answer is no, that statement is safe, yielding:

```
null
```

However, in general, the concern is valid: you need to make sure that what you pass a nullable type to can support `null`. However, that is the big advantage of this being part of Kotlin's type system: most of the time, if `null` is not allowed, a nullable type is not allowed, and you fail with some form of compilation error. If Kotlin code allows you to pass a `null` — as `println()` does — then it is up to that code to handle `null` gracefully.

Expressions with Nullable Types

Of course, there will be times when we have to cope with `null`. We want to do certain things if the value is not `null`, and do something else if the value is `null`. There are a variety of things in Kotlin that can help you work with `null` values.

Intrinsically Safe Stuff

There are many things in Kotlin, supplied by the language or its standard library of classes and functions, that support `null`. This includes things that you might not expect.

For example, `CharSequence?` (from which `String?` inherits) has an `isEmpty()` function. As one might expect, it returns `true` if the value is `null` or has no characters (e.g., it is the empty string, `""`). So, this works:

```
fun foo(message: String?) {
    println(message.isEmpty())
}

fun main() {
    foo("Hello, world!")
    foo("")
    foo(null)
}
```

(from "[Functions on Nullable Types](#)" in the *Klassbook*)

We get:

```
false
true
true
```

Not everything is allowed — many functions are defined on the core non-nullable type, rather than on its nullable counterpart.

So, for example, `Int` has a `dec()` function that returns the value decremented by one. That is defined on `Int`, not `Int?`, and so this code does not work:

```
val one = 1
println(one.dec())

val maybeZero : Int? = null
println(maybeZero.dec())
```

But, since this is a compile-time error, there is no harm in trying! And the compile error hints at a couple of solutions to the problem:

NULLABILITY

```
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable
receiver of type Int?
println(maybeZero.dec())
           ^
```

Safe Calls

Normally, to call a function on an object, you use a dot notation (.).

One option for calling a function on a variable, parameter, or property that is of a nullable type is to use the safe-call operator (?.). Then, one of two things will happen:

- If the value is null, your function call is ignored, and null is the result
- If the value is not null, your function call is made as normal

So, this works:

```
val one : Int? = 1
println(one?.dec())

val maybeZero : Int? = null
println(maybeZero?.dec())
```

(from "[Safe Calls](#)" in the *Klassbook*)

This prints:

```
0
null
```

So, the first `dec()` call happens as normal, because `one` is not null, and the second `dec()` call is replaced by `null`, as `maybeZero` is null.

The type of the result of a `?.` function call is a nullable edition of whatever type the function call normally returns. Calling `dec()` on an `Int` returns an `Int`, but calling `dec()` on an `Int` via `?.` returns an `Int?`.

This means that you can chain `?.` calls:

NULLABILITY

```
val three : Int? = 3
println(three?.dec()?.dec()?.dec())

val maybeZero : Int? = null
println(maybeZero?.dec()?.dec()?.dec())
```

(from "[Safe Call Chains](#)" in the *Klassbook*)

This prints the same result:

```
0
null
```

The Elvis Operator

`?.` function calls behave differently depending on whether the “receiver” (the object on which you are calling the function) is `null` or not.

Similarly, the “Elvis operator” — `?:` — returns different values depending on whether the left-hand side of the operation is `null` or not:

- If the left-hand side is not `null`, the Elvis operator returns the left-hand value
- If the left-hand side is `null`, the Elvis operator returns the right-hand value

```
val one : Int? = 1
println(one ?: "um, this should not be printed")

val maybeZero : Int? = null
println(maybeZero ?: "Elvis has not left the building")
```

(from "[The Elvis Operator](#)" in the *Klassbook*)

This prints:

```
1
Elvis has not left the building
```

In the first `println()` call, `one` is not `null`, so we print the value of `one`. In the second `println()` call, `maybeZero` is `null`, so we print the string that appears to the right of the operator.

NULLABILITY

Note that `return` and `throw` are valid things to have on the right-hand side of an Elvis operator:

```
fun printOrNull(value: Int?) {
    val nonNullable = value ?: return

    println(nonNullable)
}

fun main() {
    val one : Int? = 1

    printOrNull(one)

    val maybeZero : Int? = null

    printOrNull(maybeZero)
}
```

(from "[Elvis Returns!](#)" in the *Klassbook*)

Here, we only get one line of output:

```
1
```

In the case where we call `printOrNull()` with a `null` value, the first line of the function will return, bypassing the `println()` call.

But now, a quick FAQ:

Why Is This Called the “Elvis Operator”?

If you turn your head to the side and look at the `?:` operator, it looks a bit like a pair of eyes, above which is a [pompadour hairstyle](#). Elvis Presley famously wore his hair in a pompadour in his early career.

Who is Elvis Presley?

Ask your parents.

My Parents Are Asking: Who is Elvis Presley?

[Elvis Presley](#) was an American rock-and-roll icon of the 1950's through 1970's.

Isn't This FAQ a Bit of a Sidetrack for a Programming Book?

[Don't be cruel.](#)

Null Checks and Smart Contracts

Kotlin's compiler can help avoid some of the pain of dealing with nullable types. If the compiler knows that something cannot be null, due to some prior check, it relaxes the rules requiring safe calls (e.g., `?.`).

For example, let's see if our `Int?` is even, odd, or null:

```
fun evenOrOddOrNull(value: Int?) {
    if (value != null) {
        if (value.rem(2) == 0) {
            println("Even!")
        }
        else {
            println("Odd!")
        }
    }
    else {
        println("Null!")
    }
}

fun main() {
    val one : Int? = 1

    evenOrOddOrNull(one)

    val maybeZero : Int? = null

    evenOrOddOrNull(maybeZero)
}
```

(from "[Smart Contracts](#)" in the [Klassbook](#))

`rem()` is a function on `Int` that takes another `Int`, divides the two, and returns the remainder. So, `rem(2)` will return 0 for even numbers and 1 for odd numbers.

This prints what you might expect:

```
Odd!
Null!
```

NULLABILITY

However, it may not be obvious why this even compiles. `value` is an `Int?`. It would seem that `value.rem(2)` should fail with the same sort of compiler error that we saw earlier in the chapter (“only safe (`?.`) or non-null asserted (`!!.`) calls are allowed on a nullable receiver of type `Int?`”).

However, we only try calling `rem()` on `value` inside of a null check (`if (value != null)`). Kotlin’s compiler knows that inside of that block, `value` cannot be null, because we just checked to see if it was null or not. Kotlin’s compiler therefore relaxes the safe-call requirement, and we can just use `.` for our `rem()` call, rather than `?.` (and have to deal with a potentially null result).

This only works when the compiler is *sure* that the value cannot be null, though. That happens quite a bit, but there will be cases when the compiler cannot be certain and defaults to the safe-call requirement:

```
fun numberizer(): Int? = 1

fun evenOrOddOrNull() {
    if (numberizer() != null) {
        if (numberizer().rem(2) == 0) {
            println("Even!")
        }
        else {
            println("Odd!")
        }
    }
    else {
        println("Null!")
    }
}

evenOrOddOrNull()
```

Here, we get our number from a `numberizer()` function. That function is declared to return an `Int?`, even though its implementation happens to always return 1. Kotlin’s compiler does not attempt to examine the implementation of `numberizer()`. It looks at the return type, sees that it is `Int?`, and assumes that it could be null. More importantly, just because `if (numberizer() != null)` succeeded and we went into the `if` block, the compiler has no guarantee that some future `numberizer()` call will return a non-null value, so it gives us a compile error for `numberizer().rem(2)`, demanding a safe call there.

Dammit, It's Not Null

Sometimes, though, you know better than the compiler. You are *sure* that a certain value is not `null`, even though the compiler thinks otherwise.

For that, there is `!!`. You can use this operator at the end of a value or expression, and it asserts to the compiler that the value or expression will not be `null`, even if from a type standpoint it could be.

The previous example is a case where you know that `numberizer()` could never return `null`, but the compiler does not. The right solution in this case would be to have `numberizer()` return `Int` instead of `Int?`. However, there will be cases where you do not have control over the return type of the function, so you cannot change it.

We can “fix” the previous example another way, via `!!`:

```
fun numberizer(): Int? = 1

fun evenOrOddOrNull() {
    if (numberizer() != null) {
        if (numberizer()!!.rem(2) == 0) {
            println("Even!")
        }
        else {
            println("Odd!")
        }
    }
    else {
        println("Null!")
    }
}

evenOrOddOrNull()
```

Here, we append `!!` to the second `numberizer()` call, to force the compiler to treat it as returning an `Int` instead of as returning an `Int?`. Hence, this snippet compiles and runs.

We also saw this in the preceding chapter, where we were forcing a `NullPointerException`:

NULLABILITY

```
var thisIsReallyNull: String? = null

println(thisIsReallyNull)
println(thisIsReallyNull!!.length)
```

(from ["Exceptions" in the Klassbook](#))

Here, we are asserting that `thisIsReallyNull` is not null via `!!`. That results in a `NullPointerException`, since `thisIsReallyNull` is really null.

`!!` is a bit of a “code smell”. Use it sparingly, as if you are ever wrong in your assertion, you will crash with a `NullPointerException` or the equivalent, as we did in the above snippet.

Nullable Types and Generics

Types that you use in generics can be nullable:

```
val listOfNullables : MutableList<String?> = mutableListOf()

listOfNullables.add("this is not null")
listOfNullables.add(null)

println(listOfNullables)
```

(from ["Nullable Generic Types" in the Klassbook](#))

Here, `listOfNullables` is a `MutableList` that can hold `String?` instead of just `String`. So, we can put both `String` objects and `null` into the list.

So, we can create a list that can contain null. What if we want the list itself to possibly be null? In that case, the `?` goes after the generic type:

```
val nullableList : MutableList<String?>? = mutableListOf()

nullableList?.add("this is not null")
nullableList?.add("this is also not null")

println(nullableList)
```

(from ["Nullable Generic Types, Continued" in the Klassbook](#))

Here, `nullableList` can either be a `MutableList` or `null`. If it is a `MutableList`, though, that list can only hold `String` objects.

NULLABILITY

If we want, we can combine the two:

```
val nullableListOfNullables : MutableList<String?>? = mutableListOf()

nullableListOfNullables?.add("this is not null")
nullableListOfNullables?.add(null)

println(nullableListOfNullables)
```

So now not only can `nullableListOfNullables` itself be null, but if it is an actual `MutableList`, the list can hold null values, in addition to `String` objects.

And if you find all of this confusing... well, perhaps that is another reason why `?` was chosen as the symbol to use for nullability.

Nullable Types and Casts

When it comes to casts and nullable types, you can use the nullable form of `as` (`as?`):

```
val nullableList : MutableList<String>? = mutableListOf()

nullableList?.add("this is not null")
nullableList?.add("this is also not null")

val simplerList = nullableList as? List<String>

println(simplerList)
```

(from ["Nullable Casts" in the Klassbook](#))

In this case, `simplerList` winds up being a `List?` of `String` objects. If the object being cast is null, you wind up with a null value.

`as?` is also useful for cases where the object might not be of the desired type:

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val kermit : Animal = Frog()

    val notAnAxolotl = kermit as? Axolotl
}
```



```
println(notAnAxolotl)
}
```

(from ["Nullable Casts, Continued" in the Klassbook](#))

as? results in a null value if either:

- The object being cast itself is null, or
- The object being cast is not of the type that you are trying to cast it to

In this case, `kermit` is not an `Axolotl`. As a result, `notAnAxolotl` winds up being null, with the variable being typed as `Axolotl?`.

Objective: Minimize Nulls

Despite all of these things, null still winds up being a pain. To a degree, that is intentional. By making working with null annoying, the Kotlin language designers are trying to steer you to minimize your use of null values. Yet, at the same time, you can still work with libraries and frameworks for which null is a possibility (e.g., Java libraries used by Kotlin/JVM projects).

In particular, one common use of null that you can try to eliminate is using it as some “magic” or default value. In classic programming, we often use null to signify “we do not have this value yet” or “there is no value for this property”. In Kotlin, we can use things like sealed classes to avoid the null. So, instead of:

```
data class Customer(val name: String)

data class Order(val customer: Customer? = null)

val order = Order()
```

...you could have:

```
sealed class OrderingEntity {
    data class Customer(val name: String) : OrderingEntity()
    object Unassigned : OrderingEntity()
}

data class Order(val customer: OrderingEntity = OrderingEntity.Unassigned)

val order = Order()
```

NULLABILITY

This is somewhat more verbose. However, it more correctly models the situation (we do not have a customer yet, so it is unassigned). And, we avoid any possible `NullPointerException` or having to deal with lots of `?.` or `?:` and the like to make use of the customer property.

Scope Functions

As we saw [in an earlier chapter](#), “scope” refers to where variables can be referenced. For example, a function creates a scope; variables declared in that function are not accessible outside of that function.

Similarly, variables declared inside of a lambda expression are not accessible outside of that lambda expression. However, such lambda expressions need to be applied to something, such as being used by `forEach()`.

Kotlin has six global functions that allow you to create a lambda expression to declare a limited scope inside of a function.

That may sound both esoteric (why would one care?) and repetitious (do we really need six of these? can’t we use just one?).

In truth, these “scope functions” get used a lot in common Kotlin development, so it is important to understand what they are and how they get used.

let()

In [the preceding chapter](#), we saw lots of ways of dealing with values that might potentially be null, because they are referenced by variables or properties with nullable types.

Another popular way of dealing with nullable types is to use the first of the scope functions: `let()`:

```
val one = 1

one.let { println(it.dec()) }
```

SCOPE FUNCTIONS

```
val maybeZero : Int? = null

maybeZero?.let { println(it.dec()) }
```

(from ["let\(\)" in the Klassbook](#))

The `let()` function can be called on any object. It takes a lambda expression, and it passes whatever `let()` was called on into the lambda expression as a parameter. We can refer to that parameter as `it`, as in the above examples, or explicitly name it (e.g., `one.let { value -> println(value.dec()) }`).

This may seem silly. And, in the case of the first `let()`, where we are calling it on `one`, it probably is silly.

The second scenario, though, is useful. Because we are using a safe call (`?.`), `let()` is only called when `maybeZero` is not `null`. Otherwise, the `let()` call is skipped. As a result, it is an `Int`, not an `Int?`, because the Kotlin compiler knows that by definition the parameter passed to the lambda expression cannot be `null` — otherwise, we would have not called `let()` in the first place. So while `maybeZero` has to be referenced using safe calls and related techniques, code *inside the lambda expression* does not have to worry about that. In effect, we have “de-null-ed” the value. But, in our case, `maybeZero` is `null`, so that `let()` is skipped, and we only print 0 for the first `let()`.

This is great for cases where you have a bunch of work that you want to perform on a value if it is not `null`, and you are happy to skip over that work when it is `null`.

The `let()` function returns whatever the last statement is inside the lambda expression, and we can use that to populate a variable, as a parameter to a function, or whatever:

```
val one = 1

println(one.let { it.dec() })

val maybeZero : Int? = null

println(maybeZero?.let { it.dec() })
```

(from ["let\(\) Return Value" in the Klassbook](#))

Here, we only do the `dec()` call inside of the lambda expression, and we print whatever `let()` returns. This results in:

```
0  
null
```

So, in the maybeZero scenario, ?. sees that maybeZero is null and skips the let() call, returning null, and so we print null to the output.

In summary, let():

- Is called on some object
- Takes whatever you call it on and passes that into the lambda expression as a parameter
- Returns whatever the last statement of the lambda expression evaluates to

apply()

Most programmers have run into cases where they had to repeat a reference to a variable or something a lot:

```
import java.util.Calendar  
  
val sometime = Calendar.getInstance()  
  
sometime.set(Calendar.YEAR, 1980)  
sometime.set(Calendar.MONTH, 1)  
sometime.set(Calendar.DAY_OF_MONTH, 22)  
sometime.set(Calendar.HOUR_OF_DAY, 17)  
sometime.set(Calendar.MINUTE, 0)  
sometime.set(Calendar.SECOND, 0)  
sometime.set(Calendar.MILLISECOND, 0)  
  
println(sometime)
```

You have some object and you need to call a whole bunch of methods on it to configure it, such as setting the individual fields of a Calendar object, as we are doing here.

We could simplify this with let():

```
import java.util.Calendar  
  
val sometime = Calendar.getInstance()  
  
sometime.let {  
    it.set(Calendar.YEAR, 1980)  
}
```

SCOPE FUNCTIONS

```
it.set(Calendar.MONTH, 1)
it.set(Calendar.DAY_OF_MONTH, 22)
it.set(Calendar.HOUR_OF_DAY, 17)
it.set(Calendar.MINUTE, 0)
it.set(Calendar.SECOND, 0)
it.set(Calendar.MILLISECOND, 0)
}

println(sometime)
```

However, `apply()` works even better for this scenario. You call it on some object, supplying a lambda expression. The object becomes `this` inside of the lambda expression, allowing you to just call functions on it without having to name it.

Alas, Kotlin/JS does not have `java.util.Calendar`, so we cannot use that in the Klassbook. So, let's look at an alternative construct:

```
data class IntPropertyBag(private val pieces: MutableMap<String, Int> = mutableMapOf()) {
    fun set(key: String, value: Int) {
        pieces[key] = value
    }
}

fun main() {
    val sometime = IntPropertyBag()

    sometime.apply {
        set("ID", 330258648)
        set("YEAR", 1979)
        set("HOW_MANY_ROADS_MUST_A_MAN_WALK_DOWN", 42)
    }

    println(sometime)
}
```

(from "[apply\(\)](#)" in the Klassbook)

Here, pretend that `IntPropertyBag` is some truly awful class that you are getting from some third-party library. You have no choice but to use it, despite the fact that its API is fairly limited: you can give it one named integer at a time via `set()`. Here, rather than spell out the `sometime`. part of calling `set()` three times to set three integer properties, we use `apply()` to make this be the bag, so we can call `set()` without referencing the bag directly.

Also, `apply()` returns whatever object that you called it on, so we can further simplify this as:

SCOPE FUNCTIONS

```
data class IntPropertyBag(private val pieces: MutableMap<String, Int> = mutableMapOf()) {
    fun set(key: String, value: Int) {
        pieces[key] = value
    }
}

fun main() {
    val ultimateStuff = IntPropertyBag().apply {
        set("ID", 330258648)
        set("YEAR", 1979)
        set("HOW_MANY_ROADS_MUST_A_MAN_WALK_DOWN", 42)
    }

    println(ultimateStuff)
}
```

(from "[apply\(\) Return Value](#)" in the *Klassbook*)

This is the same as the previous example, except that we are just chaining `apply()` right onto the call to the `IntPropertyBag` constructor.

Of course, we could also put the `println()` in here:

```
val ultimateStuff = IntPropertyBag().apply {
    set("ID", 330258648)
    set("YEAR", 1979)
    set("HOW_MANY_ROADS_MUST_A_MAN_WALK_DOWN", 42)

    println(this)
}
```

Or, we could eliminate `ultimateStuff` entirely:

```
println(IntPropertyBag().apply {
    set("ID", 330258648)
    set("YEAR", 1979)
    set("HOW_MANY_ROADS_MUST_A_MAN_WALK_DOWN", 42)
})
```

In summary, `apply()`:

- Is called on some object
- Takes whatever you call it on and makes it be the “current” object — this — in the scope of the lambda expression
- Returns whatever object you called it on

run()

Some of the other scope functions simply remix features of `apply()` and `let()` a bit differently. `run()`, for example, works a bit like `apply()`, in that whatever you call `run()` on becomes this in the scope of the lambda expression that you supply. However, `run()` also works a bit like `let()`, in that the result of `run` is whatever the last statement of the lambda expression returns.

`run` is useful when you want to call a bunch of functions on some object, but then the end result is not the object itself, but something else... such as the result of one of those functions:

```
data class IntPropertyBag(private val pieces: MutableMap<String, Int> = mutableMapOf()) {
    fun set(key: String, value: Int) {
        pieces[key] = value
    }
}

fun main() {
    val ultimateStuff = IntPropertyBag().run {
        set("ID", 330258648)
        set("YEAR", 1979)
        set("HOW_MANY_ROADS_MUST_A_MAN_WALK_DOWN", 42)

        toString()
    }

    println(ultimateStuff)
}
```

(from ["run\(\)" in the Klassbook](#))

Here, we call `run()` on a `IntPropertyBag` instance, just as we did with `apply()` in the previous section. However, `ultimateStuff` is not the `IntPropertyBag` object, but instead the result of `toString()` called on the `IntPropertyBag` object. In the context of `println()`, there is no practical difference, as `println()` will call `toString()` to generate what to print. However, in other situations, the value that you generate from the `run` lambda expression might be more distinctly different.

In summary, `run()`:

- Is called on some object
- Takes whatever you call it on and makes it be the “current” object — this — in the scope of the lambda expression
- Returns whatever the last statement of the lambda expression evaluates to

with()

`with()` works nearly identically to `run()`. Both cause an object to become the `this` inside the lambda expression, and both return the results of the last statement in that lambda expression.

The difference lies in where the object comes from:

- With `run()`, you call `run()` on the object
- With `with()`, you pass the object as a parameter:

```
data class IntPropertyBag(private val pieces: MutableMap<String, Int> = mutableMapOf()) {
    fun set(key: String, value: Int) {
        pieces[key] = value
    }
}

fun main() {
    val ultimateStuff = with(IntPropertyBag()) {
        set("ID", 330258648)
        set("YEAR", 1979)
        set("HOW_MANY_ROADS_MUST_A_MAN_WALK_DOWN", 42)

        toString()
    }

    println(ultimateStuff)
}
```

(from ["with\(\)" in the Klassbook](#))

However, this introduces a subtle change: `run()` is more flexible with nullable types than is `with()`. With `run()`, you can use `?.` to conditionally call `run()` if the object (the “receiver”) is not null:

```
var thing: String? = "foo"

val nonNullResult = thing?.run { this.isNotBlank() }

println(nonNullResult)

thing = null

val nullResult = thing?.run { this.isNotBlank() }

println(nullResult)
```

(from ["run\(\) and Safe Calls" in the Klassbook](#))

This prints:

```
true
null
```

In both cases, we are using `thing?.run { this.isNotBlank() }`, where `isNotBlank()` returns `true` if the string has one or more non-whitespace characters. In the case where `thing` is `"foo"`, `run()` is called, because `thing` is not `null`. Where `thing` is `null`, though, the `?.` skips the `run()` call, and so `nullResult` also winds up as `null`. We do not need `?.` on the `isNotBlank()` call, because the Kotlin compiler knows that the `?.run()` cannot wind up with a `null` value.

With `with()`, though, the safe-call operator (`?.`) is not an option — we pass the possibly-null value as the parameter. Our lambda expression would need to be able to deal with the possibly-null value.

In summary, `with()`:

- Is *passed* some object
- Takes whatever you call it on and makes it be the “current” object — this — in the scope of the lambda expression
- Returns whatever the last statement of the lambda expression evaluates to

also()

Like `run()`, `also()` is a bit of a “mash-up” of `let()` and `apply()` features. Like `let()`, the object you call `also()` on is made available as a parameter to your lambda expression, so you can refer to it as `it` or some custom name. Like `apply()`, though, `also()` returns the object you call it on.

```
data class IntPropertyBag(private val pieces: MutableMap<String, Int> = mutableMapOf()) {
    fun set(key: String, value: Int) {
        pieces[key] = value
    }
}

fun main() {
    val ultimateStuff = IntPropertyBag().also {
        it.set("ID", 330258648)
        it.set("YEAR", 1979)
        it.set("HOW_MANY_ROADS_MUST_A_MAN_WALK_DOWN", 42)
    }

    println(ultimateStuff)
}
```

SCOPE FUNCTIONS

(from "[also\(\)](#)" in the *Klassbook*)

In summary, `also()`:

- Is called on some object
- Takes whatever you call it on and passes that into the lambda expression as a parameter
- Returns whatever object you called it on

`use()`

`use()` looks and works a lot like `let()`, with two major differences.

First, you cannot call `use()` on any object. It has to be something that implements the Java `Closeable` interface. This interface, added in Java 5, represents some resource that can be closed, such as a stream or socket.

Second, after executing your block of code, `use()` calls `close()` on that `Closeable`. It will do this even if your block throws an exception, by means of using a `try/finally` structure. Hence, by using `use()`, you know that whatever this resource is will be properly closed.

In summary, `use()`:

- Is called on some object that implements the `Closeable` interface
- Takes whatever you call it on and passes that into the lambda expression as a parameter
- Returns whatever the last statement of the lambda expression evaluates to
- Calls `close()` on the `Closeable` object

Summary

To recap, here are the differences among the six scope functions covered here:

SCOPE FUNCTIONS

Function	How You Call It	How You Reference the Object	What It Returns
let()	Call on an object	it	block result
apply()	Call on an object	this	the object
run()	Call on an object	this	block result
with()	Pass object as parameter	this	block result
also()	Call on an object	it	the object
use()	Call on a Closeable	it	block result

Functional Programming

Object-oriented (OO) programming has been used in “serious work” for decades. Many of the most popular languages — Java, C++, C#, Objective-C, Swift, Ruby — are based on object-oriented programming. Other languages, such as JavaScript, also leverage objects, but using somewhat different approaches (“prototype-based” vs. “class-based” object-oriented programming).

It is easy to forget that there are other ways to write a programming language.

One alternative is functional programming. There are plenty of languages, such as Clojure, Lisp, and Scheme, that have their foundations in functional programming. These languages are not nearly as popular as their OO counterparts. However, elements of functional programming are being introduced into other languages, blending functional programming with OO. Kotlin is one such language.

In this chapter, we will explore what functional programming means and how you work with it in Kotlin.

Your App Might Not Be Functional

For ordinary people who speak English, “functional” means “it works as expected”. As such, somebody telling you that your app is not functional might be a bit of an insult.

However, computer programmers might well mean “functional” in the context of functional programming. In that case, most apps are not functional, as functional programming has been a relatively niche technique, though one that is gaining mainstream momentum.

As the name suggests, functional programming's primary unit of code is a function, just as object-oriented programming's primary unit of code is an object. As with other forms of programming, a function in functional programming takes some input, performs some work, and returns some output.

The ideal type of function in functional programming is a “pure” function. A pure function has no side effects, such as changing the contents of a database. If you call a pure function one million times with the same input, you should get the same output each of those million times.

Functional programming also emphasizes function composition, using “higher-order” functions. For example, you might have a `sort()` function that can sort some collection of data. However, `sort()` itself needs to be supplied not only with the data but with some other function that knows how to compare two data elements to determine their order. `sort()`, therefore, might take two parameters:

- The data to be sorted
- Some identifier or reference to the function to use for order comparison

In this case, `sort()` is considered to be a “higher-order” function, as it takes another function as a parameter to help define the overall work to be performed.

When non-functional programming languages start to introduce functional techniques, it is often with an eye towards stream processing. Here, a “stream” could be:

- A collection of objects
- A series of events, such as user input events or database change events
- A literal “stream”, usually referring to some source of bytes of data, such as a file or socket

That is how Kotlin approaches functional programming: making it easy to use functional techniques in certain situations, without necessarily making functional programming the one-and-only approach offered by the language.

Where Immutability Comes Into Play

Immutability is often a key feature of functional programming languages. Immutability helps to ensure that impure functions do not break pure functions by changing data “behind the back” of the pure functions.

For example, there is a `first()` function available for us on `List` that returns the first element in the list that matches some rule provided via a lambda expression:

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    val leetEvent = events.first { it.id == 1337 }

    println(leetEvent)
}
```

(from "[first\(\)](#)" in the *Klassbook*)

We will explore `first()` in greater detail [later in this chapter](#). It is an example of functional programming in Kotlin, where `first()` is a pure function that acts on its input (a `List`) and return its output (the first element for which the lambda expression returns true).

However, while `first()` is pure — because the authors of Kotlin ensured that it was — it is possible that *the lambda expression* might be impure and have side effects. In particular, we might get strange results if:

- The lambda expression could modify the `Event` that it was comparing
- The lambda expression could modify the `events` list while we were trying to find something inside of it

As a result, we try to use immutable objects where possible, to reduce the likelihood that we will make those sorts of mistakes.

This also helps a lot with parallel programming, where we are performing operations in parallel across multiple threads. Developers who have had to deal with thread safety before know full well how nasty it can be to track down “timing bugs” where changes from one thread affect another thread’s work. Pure functions and immutable objects make it easier to ensure thread safety, as one thread cannot readily affect another thread’s work.

Examples of Functional Kotlin

A lot of higher-order functions are available for lists and arrays, though some are available on other types, including one that we will see [later in this chapter](#). We get lists all the time, from database queries to Web service calls. Kotlin’s higher-order functions make it easy to manipulate those lists.

For those of you with RxJava experience, a lot of these will look familiar. RxJava attempts to implement a functional API on top of Java for processing streams of data. RxJava operators are akin to higher-order functions, particularly when you use them on Java 8+ and can use lambda expressions. However, while RxJava is focused on processing streams asynchronously, Kotlin does not address threading directly with its higher-order functions.

We covered some of these back in [the chapter on collections](#).

`first()` / `firstOrNull()`

Kotlin offers a really simple `first()` function that returns the first element from the list:

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    println(events.first())
}
```

(from "[first\(\) Sans Lambda](#)" in the *Klassbook*)

However, the more powerful form of `first()` takes a lambda expression and returns the first element that matches, as we saw above:

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    val leetEvent = events.first { it.id == 1337 }

    println(leetEvent)
}
```

(from "[first\(\)](#)" in the *Klassbook*)

`first()` will throw a `NoSuchElementException`, though, if there is no matching element. So unless you are in position to catch that exception or know for certain that there will always be a match, `first()` is annoying.

`firstOrNull()` works the same as `first()`, but it returns `null` if there is no match:

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))
}
```

```
val noEvent = events.firstOrNull { it.id == 2343 }
println(noEvent)
}
```

(from ["firstOrNull\(\)" in the Klassbook](#))

With the Elvis operator, this also makes it easy for you to implement a first-or-default pattern:

```
data class Event(val id: Int)
fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))
    val oneEvent = events.firstOrNull { it.id == 2343 } ?: Event(2343)
    println(oneEvent)
}
```

(from ["firstOrNull\(\) and the Elvis Operator" in the Klassbook](#))

filter()

Whereas `first()` and `firstOrNull()` only give you one element (at most), `filter()` gives you another list or array with the subset matching some business rule (i.e., where a lambda expression returns true), as we saw back in the chapter on collections:

```
val things = listOf("foo", "bar", "goo")
things.filter { it[1]=='o' }.forEach { println(it) }
```

(from ["filter\(\)" in the Klassbook](#))

Higher-order functions are designed to be chained together, so you can use `filter()` and `firstOrNull()` to find the first event with a negative even ID:

```
data class Event(val id: Int)
fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))
    val evenNegativeEvent = events.filter { it.id % 2 == 0 }.firstOrNull { it.id < 0 }
    println(evenNegativeEvent)
}
```

(from ["Chaining Higher-Order Functions" in the Klassbook](#))

Here, our `filter()` lambda expression returns those whose IDs are even (the modulo of 2 equals 0). Our `first()` lambda expression then finds the first of those where the ID is negative.

In this case, `firstOrNull()` could just implement both rules, though, using the logical-and operator (`&&`):

```
data class Event(val id: Int)

val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42),
Event(-6))

val evenNegativeEvent = events.firstOrNull { it.id % 2 == 0 && it.id < 0 }

println(evenNegativeEvent)
```

map()

Sometimes, we have data in one form that we need to convert into another form. For that, there is the `map()` operator, as we saw in the chapter on collections:

```
val things = listOf("foo", "bar", "goo")

things
  .map { it.toUpperCase() }
  .forEach { println(it) }
```

(from "[map\(\)](#)" in the *Klassbook*)

`map()` takes a lambda expression, passes each element in the stream to that lambda expression, and collects the objects returned by the lambda expression.

any()

Sometimes, we want to run an algorithm on a collection and get some single result back. For example, `any()` returns true if any of the elements in the collection result in a true value being returned from a supplied lambda expression:

```
data class Event(val id: Int)

fun main() {
  val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

  println(events.any { it.id < 0 })
  println(events.any { it.id > 100000 })
}
```

(from "[any\(\)](#)" in the Klassbook)

Here, the first `any()` returns `true`, because one of the elements does have a negative ID value. The second `any()` returns `false`, because none of the elements has an ID above 100000.

Note that `any()` only tests elements until it gets a `true` result (or reaches the end, whichever comes first):

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    println(events.any {
        println(it)

        it.id > 10
    })
}
```

(from "[any\(\) and Short Circuits](#)" in the Klassbook)

Here, we introduce a side-effect into our lambda expression: printing the current item being examined by `any()`. Running this gives us:

```
Event(id=1)
Event(id=5)
Event(id=1337)
true
```

Once `any()` gets a `true` value, it stops and returns `true`, so we do not wind up with printed output of the other events in the list.

fold()

These sorts of higher-order functions do not have to be as simple as the ones shown above. They can be as complex as is necessary, and that in turn might require more complex lambda expressions.

For example, sometimes you want to perform a calculation on each of the elements in a collection and generate a single result from those. For that, we have `fold()`. `fold()` takes two parameters:

- An initial value
- A lambda expression that takes an “accumulator” value and an element from

FUNCTIONAL PROGRAMMING

the collection and returns a new value based on those two

The first invocation of the lambda expression gets the initial value as the accumulator value. The second invocation gets the result of the first invocation as the accumulator value. And so on through the collection:

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    val joined = events.fold("") { str, event ->
        if (str.length==0) event.toString() else "$str,$event"
    }

    println(joined)
}
```

(from "[fold\(\)](#)" in the *Klassbook*)

Each time our lambda expression gets invoked, it gets the concatenated results so far plus the next event in the list. Our lambda expression sees if the concatenated results are empty, and if so just returns a String representation of the first event. Otherwise, it adds a comma and the String representation of the current event. As a result, that accumulated String keeps growing:

Invocation	str Value	event Value
1	""	Event(1)
2	"Event(1)"	Event(5)
3	"Event(1),Event(5)"	Event(1337)
4	"Event(1),Event(5),Event(1337)"	Event(24601)
5	"Event(1),Event(5),Event(1337),Event(24601)"	Event(42)
6	"Event(1),Event(5),Event(1337),Event(24601),Event(42)"	Event(-6)

When `fold()` reaches the end of the collection, it returns whatever the last lambda expression returned.

Of course, Kotlin offers a `joinToString()` function that would handle this for us more simply:

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    val joined = events.joinToString(",")

    println(joined)
}
```

(from ["joinToString\(\)" in the Klassbook](#))

Function Types

These might look like magic, or at least like the sort of thing that can only be provided as part of a core language implementation. In reality, these higher-order functions are not that complicated, but they take advantage of a key feature of Kotlin: function types. A function type allows you to pass what looks like a function as a parameter to another function.

For example, the `hasMatch()` function in this code snippet is a higher-order function that mimics the functionality of `any()`:

```
fun <T> hasMatch(list: List<T>, predicate: (T) -> Boolean): Boolean {
    list.forEach { item ->
        if (predicate.invoke(item)) return true
    }

    return false
}

data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    println(hasMatch(events) { it.id < 0 })
    println(hasMatch(events) { it.id > 100000 })
}
```

(from ["Function Types" in the Klassbook](#))

Functions and Generics

For an `any()`-style function, we need two things:

- The collection of items to check, and
- The lambda expression that we use to check each item until we get a match

On the surface, the collection seems easy. We could just have a parameter that is a

List or an Array or something. However, ideally, we would declare this function to be able to handle a list of any type of data. Our function does not care whether this is a list of `String` objects, a list of `Event` objects, or a list of `Axolotl` objects. All our function needs to do is to pass elements from the list to the lambda expression. So long as the list and the lambda are in agreement on types, what the type is does not matter.

`hasMatch()` uses generics not at the class level but at the function level. The `<T>` after the `fun` keyword says “hey, this function can operate on objects of some type `T`”. We can then use `T` as a placeholder for the “real” type elsewhere in the function declaration. In particular, our `list` parameter is a `List<T>`. We have no constraints on what `T` can be, so when we use `hasMatch()` a `List<Event>` works fine for a `List<T>`.

Declaring a Function Type Parameter

The second parameter to `hasMatch()` should be our lambda expression. From the standpoint of our function, though, the second parameter is a function type. `foo: String` declares a parameter named `foo` of type `String`, where the parameter name precedes the colon. Similarly, `predicate: (T) -> Boolean` declares a parameter named `predicate`. However, the type is a function type.

A function type has three elements:

- The types of the parameters to the function, enclosed in parentheses
- The arrow operator (`->`)
- The type returned by the function

In our case, we are saying that we want, as the second parameter, a function that takes in a `T` and returns a `Boolean`. `hasMatch()` does not care what the function is or does, so long as it accepts a `T` and returns a `Boolean`.

Passing a Function Type

Given that we have a higher-order function like `hasMatch()`, we need to supply a function satisfying the function parameter. There are a few ways in which we can do that.

Lambda Expressions

The most common approach, and the one used in the above example, is to pass a lambda expression.

Technically, the lambda expression ought to appear as an actual function parameter of `hasMatch()`:

```
println(hasMatch(events, { it.id < 0 }))
println(hasMatch(events, { it.id > 100000 }))
```

Here, our lambda expression is part of the comma-delimited list of parameters passed to `hasMatch()`.

However, the Kotlin compiler allows you to put the lambda expression after the closing parenthesis, if the function type parameter is the last parameter in the function's parameter list. That is why `hasMatch()` takes the list first and then the function type parameter, so we can write this instead:

```
println(hasMatch(events) { it.id < 0 })
println(hasMatch(events) { it.id > 100000 })
```

If a function takes more than one function type parameter, or if the function type parameter is not last in the list, you have to pass the lambda expression as a regular parameter, inside the parentheses of the function call. For example, if we had a `subscribe()` function that took two function type parameters for `onSuccess` and `onError`, at most we could have the second of those be outside the parentheses.

Function References

While lambda expressions are popular, they are not the only option.

The next-most common solution is to use a function reference. This is a way to identify an existing function and pass it as a parameter. If the function's signature matches what is required by the function type, then the compiler will be happy.

The most common syntax for this is to use a double-colon before the function name:

```
fun <T> hasMatch(list: List<T>, predicate: (T) -> Boolean): Boolean {
    list.forEach { item ->
        if (predicate.invoke(item)) return true
    }
}
```


FUNCTIONAL PROGRAMMING

```
    return false
}

data class Event(val id: Int)

fun lessThanZero(event: Event) = event.id < 0

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    println(hasMatch(events, ::lessThanZero))
}
```

(from ["Function Reference" in the Klassbook](#))

Here, we have a `lessThanZero()` function that takes an `Event` and returns a `Boolean`. We can use that to satisfy the function type for `hasMatch()` by using `::lessThanZero` as the second parameter to the `hasMatch()` call, replacing our lambda expression. Kotlin's type inference capability means that when we call `hasMatch(events, ::lessThanZero)`, Kotlin sees that:

- `events` is a `List<Event>`
- `lessThanZero()` takes an `Event`

This satisfies both of the generics used in the `hasMatch()` function signature, so the compiler is happy.

This syntax works for top-level functions, such as this implementation of `lessThanZero()`. It also works for member functions, if the function reference is inside the same object:

```
data class Event(val id: Int)

class Thingy {
    fun <T> hasMatch(list: List<T>, predicate: (T) -> Boolean): Boolean {
        list.forEach { item ->
            if (predicate.invoke(item)) return true
        }

        return false
    }

    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    fun lessThanZero(event: Event) = event.id < 0

    fun matchify() {
        println(hasMatch(events, ::lessThanZero))
    }
}
```

```
fun main() {
    Thingy().matchify()
}
```

(from ["Function Reference for Member Functions" in the Klassbook](#))

Here, everything is a part of a Thingy class, and so `::lessThanZero` will resolve to the `lessThanZero()` member function of Thingy. The implication is that the receiver — the object on which `lessThanZero()` will be called — is `this`, or the current instance of Thingy where we are trying to use `lessThanZero()`.

If you want to use a function on a specific receiver, put its name before the double-colon:

```
fun <T> hasMatch(list: List<T>, predicate: (T) -> Boolean): Boolean {
    list.forEach { item ->
        if (predicate.invoke(item)) return true
    }
    return false
}

data class Event(val id: Int)

class Thingy {
    fun lessThanZero(event: Event) = event.id < 0
}

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))
    val thingy = Thingy()

    println(hasMatch(events, thingy::lessThanZero))
}
```

(from ["Function References and Receivers" in the Klassbook](#))

Here, `lessThanZero()` is a function on Thingy, and thingy is an instance of Thingy, so we can use `thingy::lessThanZero` to reference a `lessThanZero()` function in the context of thingy.

Anonymous Functions

The third approach — the anonymous function — does not seem to be that widely used, so we will explore it [much later in the book](#).

Using a Function Type Parameter

You have two options for causing the function type parameter to execute and give

you a result for some input: calling `invoke()` on it or using the function type as if it were an actual function.

Calling `invoke()`

Given that you have a parameter that is a function type, to call that function, call `invoke()` on the parameter. That is what we are doing in the `hasMatch()` higher-order function:

```
fun <T> hasMatch(list: List<T>, predicate: (T) -> Boolean): Boolean {
    list.forEach { item ->
        if (predicate.invoke(item)) return true
    }

    return false
}
```

`invoke()` will take the parameters that are declared for the function type. `hasMatch()` declares that `predicate` takes a `T` instance, so we must supply a `T` instance to `invoke()`.

Similarly, `invoke()` returns whatever the function type says it should return. `predicate` is declared as returning a `Boolean`, so `invoke()` returns a `Boolean`.

Calling Directly

You can also skip the `invoke()` and treat the function type as an actual function:

```
fun <T> hasMatch(list: List<T>, predicate: (T) -> Boolean): Boolean {
    list.forEach { item ->
        if (predicate(item)) return true
    }

    return false
}

data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    println(hasMatch(events) { it.id < 0 })
    println(hasMatch(events) { it.id > 100000 })
}
```

(from "[Function Types and Direct Invocation](#)" in the *Klassbook*)

Here, we treat predicate as if it were predicate(), calling it like a function, passing the parameter and getting the result. The effect is the same as with invoke() itself:

- When called, the function type must be passed the same parameter(s) as the function type calls for
- When called, the function type returns an object of the type that the function type is declared to return

Multi-Parameter Function Types

Note that function types are not limited to a single parameter. They can support any number of parameters. For example, this collectify higher-order function uses a function type taking two parameters:

```
fun <K, V, T> collectify(input: Map<K, V>, transform: (K, V) -> T): List<T> {
    val result = mutableListOf<T>()

    for ((key, value) in input) { result.add(transform(key, value)) }

    return result.toList()
}

fun main() {
    val stuff = mapOf("foo" to "bar", "goo" to "baz")

    println(stuff)
    println(collectify(stuff) { key, value -> "${key.toUpperCase()}: ${value.toUpperCase()}" })
}
```

(from ["Function Types and Parameters" in the Klassbook](#))

The objective of collectify() is to convert the Map into a List of objects, using some supplied function type to take each key/value pair from the Map and create a corresponding object for the List.

For each entry in the Map, we invoke the transform on the key and value. We take the result of transform() and append it to a MutableList. We then return that list, converted to an immutable List via toList().

To use collectify(), we need to pass some implementation of the function type. The script shown above uses a lambda expression that returns a string made up of the uppercase renditions of the key and value.

If you run this script, you get both the original Map plus the collectify() result:

```
{foo=bar, goo=baz}
[F00: BAR, G00: BAZ]
```

In this case, though, `collectify()` is pointless, as `Map` already supports the unfortunately-named `map()` function, which does the same thing. However, `map()` uses `Entry` objects instead of the keys and values for the function type:

```
val stuff = mapOf("foo" to "bar", "goo" to "baz")

println(stuff)

val mappedMap = stuff.map { entry -> "${entry.key.toUpperCase()}:
${entry.value.toUpperCase()}" }

println(mappedMap)
```

So, our lambda expression gets an `Entry` and needs to retrieve the key and value from it.

Scope Functions and Function Types

Most places that you see something like a lambda expression, there is a function with a function type parameter that invokes that lambda expression.

For example, previously we examined [Kotlin's scope functions](#), such as `let()` and `apply()`. Those are implemented using function type parameters.

However, if you look at the Kotlin documentation for these, they will look a bit strange. For example, this is [the declaration for `let\(\)`](#):

```
inline fun <T, R> T.let(block: (T) -> R): R
```

`T` and `R` are generic types, and `block` is a function type that accepts a `T` parameter and returns an `R`. However, there are two elements of this declaration that we have not covered yet.

First is `inline`. That is an optimization and is optional, so we will cover it [much later in the book](#).

The other part is `T.let`. `let` is the function name, but what is the `T`?

This is the way that you declare an “extension function”, where we can add functions to existing types, even for classes and stuff that we did not create. Extension

functions are a powerful — and somewhat dangerous — capability in Kotlin, and we will explore them in [an upcoming chapter](#).

Type Aliases

One problem with function types is that they are a bit long, particularly if you have a bunch of parameters to declare. Something like $(K, V) \rightarrow T$ is not bad, because we tend to use single-letter identifiers for generics. But, this could easily become $(String, Restaurant) \rightarrow CustomerOrder$ or something like that, which is a long data type.

Even then, it may not be bad, if you only use that function type in one place. However, if you use that function type in lots of places, the length gets magnified. Plus, it becomes a lot more work to change them, if you need to replace `Restaurant` with something else (e.g., `FoodProvider`).

One way to help manage this is with type aliases. As the name suggests, type aliases allow you to come up with your own “shorthand” identifier that maps to some other type. You can use type aliases for any sort of data type, but they are particularly useful with function types.

Declaring a type alias is very easy: use `typealias`, followed by your desired alias, `=`, and the type that the alias maps to:

```
typealias StringMap = Map<String, String>
typealias StringPair = Pair<String, String>
typealias Pairmonger = (String, String) -> StringPair
```

A type alias can even reference another type alias. Here, the `Pairmonger` type alias is defined in terms of `StringPair`, which itself is a type alias.

You can then use those aliases wherever you might use the actual type:

```
typealias StringMap = Map<String, String>
typealias StringPair = Pair<String, String>
typealias Pairmonger = (String, String) -> StringPair

fun collectify(input: StringMap, transform: Pairmonger): List<StringPair> {
    val result = mutableListOf<StringPair>()

    for ((key, value) in input) { result.add(transform(key, value)) }

    return result.toList()
}
```

```
}  
  
fun main() {  
    val stuff = mapOf("foo" to "bar", "goo" to "baz")  
  
    println(stuff)  
    println(collectify(stuff) { key, value -> key to value } )  
}
```

(from ["Type Aliases" in the Klassbook](#))

Using too many aliases will make it difficult for newcomers to understand your code, as they require a level of “mental indirection” that will take time to learn. Try to limit your aliases to where the alias is significantly simpler than the type being aliased. In the example above, `StringMap` and `StringPair` would not be worth aliasing in most projects, while `PairMonger` may be more reasonable.

Arrays, Collections,... And Sequences

`map()`, `fold()`, and similar higher-order functions in Kotlin can be used with arrays and many types of collections, such as `List`.

They can also be used with `Sequence`. Unlike `List` — which has its origins in the Java `java.util.List` type — `Sequence` is a Kotlin-specific type. On the surface it can be used a lot like a `List`, particularly in functional programming. In reality, `Sequence` is a substantially different construct.

What’s a Sequence?

Technically, `Sequence` is more comparable with Java’s `Iterable`. Both interfaces have an `iterator()` method that returns some `Iterator` to iterate over stuff. In Java, all of the one-dimensional collection classes that you are used to, like `ArrayList` and `HashSet`, have implementations of `Iterable`.

The difference lies in the implied contract. The assumption is that an `Iterable` represents an actual collection of stuff, where that stuff exists in memory and can be iterated over. As a result, `Iterable` also has methods like `forEach()` that take advantage of this concrete collection. `Sequence`, by contrast, is *lazy*, meaning that there should be no assumption that the data over which you are iterating actually exists prior to being handed to the iterator.

Ummm... Why Bother?

Kotlin's developers designed its higher-order functions that work on Sequence to be item-at-a-time, whereas they needed to make their higher-order functions that work on Iterable to be collection-at-a-time.

Let's turn back to an example from earlier in the chapter:

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    val evenNegativeEvent = events.filter { it.id % 2 == 0 }.firstOrNull { it.id < 0 }

    println(evenNegativeEvent)
}
```

(from "[Chaining Higher-Order Functions](#)" in the *Klassbook*)

Here, we chain two higher-order functions: `filter()` and `firstOrNull()`. These are being applied to a `List`, which implements `Iterable`. As a result, what happens is:

- We create a `List` of six `Event` objects
- `filter()` creates a `List` of two `Event` objects (those with an ID that is even)
- `firstOrNull()` returns a single `Event`

Let's change this a bit, by adding a `map()` operation and changing the order of the events so that the `-6` one is second:

```
val events = listOf(Event(1), Event(-6), Event(5), Event(1337), Event(24601), Event(42))

val evenNegativeEvent = events
    .map { it.id }
    .filter { it % 2 == 0 }
    .firstOrNull { it < 0 }

println(evenNegativeEvent)
```

Now, the flow is:

- We create a `List` of six `Event` objects
- `map()` creates a `List` of six `Integer` objects
- `filter()` creates a `List` of two `Integer` objects (those that are even)
- `firstOrNull()` returns a single `Integer`

For a starter list of six objects, that is not too bad. But imagine that the list had 6,000 objects. Now, we are allocating memory for the 6,000-item initial list, the 6,000-item list from `map()`, plus some smaller list from `filter()`.

With a Sequence, each item is processed through the entire chain on an item-by-item basis. With that, our flow would be:

- We create a Sequence of six Event objects
- `map()` emits the first Integer (1)
- `filter()` skips that item, as it is not even
- `map()` emits the second Integer (-6)
- `filter()` emits -6, since it is even
- `firstOrNull()` returns -6, since it is negative
- ...and we are done

This is much more memory efficient, as we are not creating intermediate List objects. Those gains can be quite significant for large lists and complex chains of higher-order functions.

How Do I Get a Sequence?

There are many ways to get your hands on a Sequence, either by creating one from scratch or getting one from something else.

`sequenceOf()`

Just as we have `arrayOf()`, `listOf()`, and so forth, we have a `sequenceOf()` global function in Kotlin to create a Sequence from a set of items known at compile time.

So, for example, the Sequence scenario outlined above looks like:

```
data class Event(val id: Int)

fun main() {
    val events = sequenceOf(Event(1), Event(-6), Event(5), Event(1337), Event(24601), Event(42))

    val evenNegativeEvent = events
        .map { it.id }
        .filter { it % 2 == 0 }
        .firstOrNull { it < 0 }

    println(evenNegativeEvent)
}
```

(from "[Sequences and sequenceOf\(\)](#)" in the *Klassbook*)

asSequence()

If you already have a `List` or `Array`, you can call `asSequence()` on it to get a `Sequence` based on the `List` or `Array` contents. This may be useful if some code other than yours is creating the `List` or `Array`, so you cannot start with a `Sequence` yourself.

generateSequence()

The `generateSequence()` global function in Kotlin creates a `Sequence` that is wrapped around a function type (e.g., lambda expression) that you supply. Where that function type gets its data from is up to you. Your function will be called as each item in the `Sequence` is needed, until such time as your function returns `null` to signal that you are out of data.

```
import kotlin.random.Random

fun percentileDice() = Random.nextInt(1,100)

fun main() {
    val sequence = generateSequence {
        percentileDice().takeIf { it < 95 }
    }

    println(sequence.toList())
}
```

(from "[generateSequence\(\)](#)" in the *Klassbook*)

The `percentileDice()` function generates a random number from 1 to 100, using `Random.nextInt()` from the Kotlin standard library. Our `generateSequence()` call then uses a lambda expression that:

- Calls `percentileDice()` to get a random number
- Uses `takeIf()` to see if it is less than 95 — `takeIf()` returns the object it is called upon if the lambda returns `true`, otherwise it returns `null`

The lambda expression ends with that `takeIf()` call, so the lambda expression returns the random number (if it is less than 95) or `null`. The result is that we have a `Sequence` of a random number of random numbers. We then use `toList()` to collect the values in a `List`, then print that `List`.

Running this might give you:

FUNCTIONAL PROGRAMMING

```
[33, 94, 19, 62, 81]
```

or:

```
[32, 84, 39, 21, 22, 51, 21, 78, 82, 3, 53, 30, 48, 64, 89, 33, 16, 28, 16, 2, 90, 38, 67, 34, 86, 71, 13]
```

or:

```
[]
```

That latter case is where the very first random number was over 95, so the Sequence terminated immediately.

Note that the `null-means-done` implementation of `generateSequence()` means that you cannot use it to generate a Sequence containing `null` objects.

Elsewhere

When working with APIs, whether from the Kotlin standard library or elsewhere, see if they have options to return a Sequence instead of a List or Array.

Extension Functions

There are lots of functions that we can call on pretty much any object. Some, like `toString()`, are the sorts of things that you might expect any object to support. Others, like many of the [scope functions](#) (e.g., `let()`, `apply()`), might seem a bit odd that you can call on anything.

Still others might not even be obvious that they are functions. Back in [the chapter on collections](#), we saw creating a Map using `mapOf()`:

```
println(mapOf("foo" to "bar", "baz" to "goo"))
```

`to()` is actually a function that we are calling on "foo" and "baz", where `to()` returns a `Pair` made up of what we are calling it on and the parameter to the function (the value after `to`).

The root class of the Kotlin class hierarchy is `Any`. All classes extend from `Any`. If you look at [the Kotlin reference to Any](#), you might expect to see lots of these functions listed. Instead, there are only three functions on `Any`:

- `equals()`
- `hashCode()`
- `toString()`

Those map to their Java equivalents in Kotlin/JVM and have other implementations in Kotlin/JS and Kotlin/Native.

So... where are all these other things, like `let()` and `apply()` and `to()` coming from, if they are not functions on `Any`?

Those are defined as extension functions, where you can provide an implementation

of a function for a class that you did not create. In this chapter, we will explore why we have these things, how we set them up, and whether they are really a good idea or not.

The Case of the Utility Function

A question often arises in object-oriented programming: “where should this function go?”

Often, the answer is obvious, as the work associated with the function is tied to some class that you are working on. Putting that function on that class is a logical move to make.

But sometimes we have functions that do not necessarily fit that pattern.

For example, suppose that you want to be able to validate a user-entered `String` to see if it looks like a valid email address. As it turns out, *lots* of really strange things can be valid email addresses, as the email address standard is very loose and forgiving. However, if you hunt around on the Web, you will find various regular expressions that can be used to validate addresses, where those regular expressions can work for most commonly-seen address structures.

Kotlin supports regular expressions via a `Regex` class. You can create one using a regular-expression pattern and, among other things, call `matches()` on it to see if a particular `String` matches the pattern:

```
val EMAIL_PATTERN = "[a-zA-Z0-9_.-]+@[a-zA-Z0-9-]+.[a-zA-Z0-9-]+(\\s)*"

fun main() {
    val emailRegex = Regex(EMAIL_PATTERN)

    println(emailRegex.matches("martians-so-do-not-exist@commonsware.com"))
    println(emailRegex.matches("this is not an email address"))
}
```

(note: this is not a particularly good regular expression for evaluating email addresses, but it is fairly short and is adequate for a book)

If we only needed this logic in one spot in the app, we could just have a function in that one spot that handles this code. But we might need to validate email addresses in a few places:

EXTENSION FUNCTIONS

- On the user registration screen, as we are using email addresses as user IDs
- On the login screen, because our user IDs are email addresses
- On the “add a friend” screen
- On the “share this content” screen
- And so on

So, where does this “is this a valid email address?” code go?

Pointless Superclasses

A classic object-oriented programming solution is to note that all four cited uses are in screens, so there should be some base class that has this function that everyone else can inherit from. For example, in Android, that could be:

- A `BaseActivity` that extends from `Activity` and has this function
- A `BaseFragment` that extends from `Fragment` and has this function
- A `BasePresenter` for some Model-View-Presenter (MVP) framework that has this function
- Or whatever

However, in general, forcing a particular inheritance model, just to provide access to some simple function, is not a good OO design approach.

Utility Classes

The other typical OO approach is to have some utility class with this function. That could be somewhat specialized, such as a `DataValidator` that knows how to perform various types of validation. Or, it could be a `Util` class that serves as a home for all sorts of wayward bits of code that have no other likely home.

This works. However, the more generic the utility class, the more likely that it becomes a “junk drawer”, gradually collecting more and more code, until you have this massive class that nobody likes to maintain.

Global Functions

Yet another approach, particularly in Kotlin, is to use a top-level function. You could just have an `isValidEmail()` function floating around somewhere that performs this validation.

This is not much of an improvement over the utility class, though. In fact, it strongly

lends itself towards “junk drawer” models, where you have some Kotlin source file with a random collection of top-level functions that needed to live somewhere.

Worse, all top-level functions share a namespace. Suppose that you have an `isValidEmail()` function, and now you add some library where *it* has its own `isValidEmail()` top-level function. Now you will have a build error, as the compiler will find duplicate functions with the same name.

Monkeying Around

Another approach that some languages have used is to allow developers to add functions to *existing classes*, including those that might be supplied as part of some standard library. That would allow you to, say, have an `isValidEmail()` function on `String` itself:

```
println("martians-so-do-not-exist@commonsware.com".isValidEmail())
println("this is not an email address".isValidEmail())
```

In the case of JavaScript, this is merely a matter of extending the prototype of the desired class:

```
String.prototype.isValidEmail = function() {
  // TODO implementation goes here
}
```

In Ruby, you can accomplish a similar thing, often referred to as “monkey-patching” a class:

```
class String
  def isValidEmail
    # TODO implementation goes here
  end
end
```

In both cases, `isValidEmail()` is added to that environment’s `String` type, so you can call `isValidEmail()` directly on a `String` object.

Kotlin supports this too, via what are known as extension functions. In fact, *lots* of stuff in Kotlin’s standard library is implemented via extension functions, instead of by implementing the functions directly on the affected types.

Declaring Extension Functions

Declaring an extension function looks and works a lot like declaring any other function. The difference is that the type the extension function is being declared upon is included as a prefix to the function name, separated by ..

```
private val EMAIL_REGEX = Regex("[a-zA-Z0-9_+-.]+@[a-zA-Z0-9-]+[.]+[a-zA-Z0-9-]+(\\s)*")

fun String.isValidEmail() = EMAIL_REGEX.matches(this)

fun main() {
    println("martians-so-do-not-exist@commonsware.com".isValidEmail())
    println("this is not an email address".isValidEmail())
}
```

(from ["Extension Functions" in the Klassbook](#))

Here, as with the JavaScript and Ruby examples, we are adding an `isValidEmail()` function to Kotlin's existing `String` type. We can then call `isValidEmail()` on candidate email addresses, whether those come from users or are hard-coded test values, such as `"martians-so-do-not-exist@commonsware.com"`.

The Syntax

The extension function behaves a lot like a regular function that you might have on the class. In particular, this is how you refer to the instance of the type. So, in `String.isValidEmail()`, this is the `String` on which you called `isValidEmail()`.

The type on which you declare the extension function can be any valid Kotlin type. This includes [nullable types](#), though the function implementation may wind up being more complex as a result:

```
private val EMAIL_REGEX = Regex("[a-zA-Z0-9_+-.]+@[a-zA-Z0-9-]+[.]+[a-zA-Z0-9-]+(\\s)*")

fun String?.isValidEmail() = this?.let { EMAIL_REGEX.matches(this) } ?: false

fun main() {
    println("martians-so-do-not-exist@commonsware.com".isValidEmail())
    println("this is not an email address".isValidEmail())
    println(null.isValidEmail())
}
```

(from ["Extension Functions on Nullable Types" in the Klassbook](#))

Here, we declare `isValidEmail()` on `String?` instead of `String`, allowing us to call `isValidEmail()` on `null` as well as on an actual `String`. However, `matches()` on a `Regex` does not accept a `null` parameter, so we use `let()` and the Elvis operator to

supply `false` as a default value for the null case.

The Location and Visibility

You can declare an extension function anywhere that you can declare a regular function. So, in the above example, the extension function is a top-level function.

You can make extension functions be `public` (the default) or `private`, as needed.

This leads to three common patterns:

- Public top-level extension functions
- Private top-level extension functions
- Private member functions

Public Top-Level

A public top-level extension function adds that function to the designated type for anything using your code. This is good for utility functions that will be used widely in the project.

Private Top-Level

A private top-level extension function is usable only within the source file that contains that function. This is good for utility functions whose implementation is fairly tightly tied to the code in the source file and would not be useful elsewhere. The private nature means that you will not collide with other private implementations in other source files.

Class Members

You can also have class members that are extension functions, limiting their access to instances of that class:

```
private val EMAIL_REGEX = Regex("[a-zA-Z0-9_.-]+@[a-zA-Z0-9-]+[.]+[a-zA-Z0-9-]+(\\s)*")

data class User(val userId: String)

class UserRegistration {
    private fun String.isValidEmail() = EMAIL_REGEX.matches(this)

    fun isValidUser(user: User) = user.userId.isValidEmail()
}
```

EXTENSION FUNCTIONS

```
fun main() {
    val registrar = UserRegistration()

    println(registrar.isValidUser(User(userId = "martians-so-do-not-exist@commonsware.com")))
    println(registrar.isValidUser(User(userId = "this is not an email address")))
}
```

(from "[Private Extension Functions](#)" in the *Klassbook*)

Here, `isValidEmail()` is an extension function on `String`, but that extension function is only accessible to instances of `UserRegistration`.

This could just as easily be written as:

```
class UserRegistration {
    private fun isValidEmail(value: String) = emailRegex.matches(value)

    fun isValidUser(user: User) = isValidEmail(user.userId)
}
```

Certainly, this second approach is more conventional. In the end, it comes down to object design. In this case, should a `String` know if it is a valid email address? Or is that really a responsibility of something else, such as `UserRegistration`?

Calling Extension Functions

Calling an extension function is indistinguishable from calling a regular function. To the caller, they look the same, and to the developer, their syntax is identical. There is no real way to look at a function invocation and determine whether it is calling a regular function or an extension function... at least from Kotlin.

Since other languages do not necessarily have this sort of capability, you may be able to tell the difference when other code calls into Kotlin code, trying to use an extension function. We will see an example of this in [an upcoming chapter on Java interoperability](#).

The Limitations

Extension functions live outside of the classes that they are extending. In the end, extension functions are “syntactic sugar” and are equivalent to a regular top-level function. So, in the case of `String.isValidEmail()`, the `isValidEmail()` extension function on `String` is really no different than a top-level `isValidEmail(value:`

EXTENSION FUNCTIONS

String) function that operated on the supplied String parameter.

As a result, extension functions have no special access to the implementation of the class being extended by the function. In particular, extension functions have no access to private or protected members.

```
class Something {  
    val thisIsPublic = true  
    private val thisIsNot = false  
}  
  
fun Something.kindGrabby() = this.thisIsPublic && this.thisIsNot
```

This code fails with a compile error:

```
error: cannot access 'thisIsNot': it is private in 'Something'  
fun Something.kindGrabby() = this.thisIsPublic && this.thisIsNot  
                                     ^
```

Since `thisIsNot` is private, `kindGrabby()` has no access to it.

Extension functions also have no ability to add new properties to the class that they are extending. As a result, extension functions have no ability to store new data. All they can do is manipulate the data that the class already stores.

Java Interoperability

There is a decent chance that you are interested in using Kotlin for Android app development, as there is a lot of Kotlin momentum in that space.

You might also be interested in using Kotlin in other places alongside Java:

- Web app development
- Desktop software development (e.g., TornadoFX)
- General tools development

Throughout the book to this point, there have been a few notes about Kotlin running on the Java Virtual Machine (JVM). Everything that you have learned so far works for Kotlin/JVM. In this chapter, we focus on things that you may have to do to get your Kotlin code to work alongside Java code, whether that is Kotlin calling into Java (e.g., the Android SDK) or Java calling into Kotlin (e.g., legacy Java code in your app).

Recap of Interoperability

As was covered [back in the first chapter](#), there are a number of layers to Kotlin with respect to environments.

A lot of Kotlin is what is considered nowadays to be Kotlin/Common. This code works on all environments that Kotlin can run on.

Kotlin then has three variants (at present) for running on specific environments:

- Kotlin/JVM for compiling Kotlin to Java bytecode for use on the JVM
- Kotlin/JS for “transpiling” Kotlin into JavaScript for use in the browser or

- with server-side engines like NodeJS (or in the Klassbook!)
- Kotlin/Native for compiling Kotlin to LLVM for use in platforms like iOS

The implementations of some things may vary based on environment. For example, [the `Regex` class](#) for working with regular expressions has different rules for Kotlin/JVM, Kotlin/JS, and Kotlin/Native. While the class always exists, the regular expression pattern language may differ, as `Regex` defers to an environment-supplied implementation.

But beyond that, you may need to add some stuff to your code base to be able to interoperate with code specific for an environment, such as Java code for Kotlin/JVM or JavaScript code for Kotlin/JS. While Kotlin aims to make it relatively painless to bridge languages, there are still differences that need to be resolved. This chapter will focus on those involving Java.

Kotlin Calling Java

Particularly for new projects, most likely you will be focused on Kotlin code calling into Java code. While Kotlin libraries are growing in number, they pale in comparison to the vast array of Java libraries. In particular, many major frameworks are implemented in Java, such as the Android SDK and Spring. While they may get some Kotlin wrappers, many developers will wind up working with them directly.

On the whole, things tend to work fairly smoothly. There are a variety of occasions where you will need to think about interoperability, but for the most part, “it just works”. This is one of the reasons why Google was comfortable in endorsing Kotlin — if having Kotlin code call into the Android SDK was going to be some huge problem, Google might have been more cautious.

In this section, we will explore some things that you will need to consider as you have your Kotlin code call out to Java classes and methods.

Java Methods and Property Syntax

If you spend time on a Kotlin/JVM project in an IDE that offers Kotlin auto-complete (e.g., Android Studio, IntelliJ IDEA), you will rapidly discover that Kotlin allows you to invoke certain Java methods as if they were Kotlin properties or other simpler forms of Kotlin syntax.

getXXX() and setXXX() Pairs

The so-called “JavaBeans” method structure in Java involves paired methods, prefixed with `get` and `set()`, with a common base and using a common type:

```
public class Something {
    private String foo;

    public String getFoo() {
        return foo;
    }

    public void setFoo(String newFoo) {
        foo = newFoo;
    }
}
```

Here, we have JavaBeans-style accessor methods for a `foo` field, with `getFoo()` returning the `foo` value and `setFoo()` changing it.

If you access this Java code from Kotlin, you can treat the `getFoo()` and `setFoo()` methods as if you were accessing a `foo` property that was public:

```
class Bar {
    fun doFoo() {
        val thingy = Something()
        val oldFoo = thingy.foo

        thingy.foo = "this is the replacement"
    }
}
```

There is nothing stopping you from calling the Java methods directly, so this works too:

```
class Bar {
    fun doFoo() {
        val thingy = Something()
        val oldFoo = thingy.getFoo()

        thingy.setFoo("this is the replacement")
    }
}
```

However, your IDE might hint to you to switch to “property access syntax”:



Figure 6: Android Studio, Suggesting Property Access Syntax

So, while calling the methods directly works, your IDE would prefer that you use the shorter syntax.

Note that property access syntax only works when there is a matching getter and setter with the same base name and operating on the same type. For example, you might have a Java class with a single getter but overloaded setters, such as with Android’s `TextView` and its `text` “property”. There are several `setText()` variants, accepting different parameters. If there is a matching pair — such as the `CharSequence` versions of `getText()` and `setText()` on `TextView` — property access syntax works:

```

class Goo {
    fun doText(tv: TextView) {
        val oldText = tv.text

        tv.text = "This is new!"
    }
}
    
```

However, other setters that work with other types require you to call the setter method directly, not use property assignment syntax:

```

class Goo {
    fun doText(tv: TextView) {
        val oldText = tv.text
    }
}
    
```

```
tv.setText(R.string.app_name)
}
}
```

If you try `tv.text = R.string.app_name`, you get a type mismatch error, as the property assignment is expecting a `CharSequence`, and `R.string.app_name` (in the world of Android) is an `Int`.

The `isXXX()` Variant

Sometimes, with `boolean` or `Boolean` values, the Java getter method uses `is...()` instead of `get...()` as a naming convention:

```
public class Something {
    private boolean foo;

    public boolean isFoo() {
        return foo;
    }

    public void setFoo(boolean newFoo) {
        foo = newFoo;
    }
}
```

This too maps to a property in Kotlin, though the property name will be the same as the getter method name — `isFoo` in this case:

```
class Bar {
    fun doFoo() {
        val thingy = Something()
        val oldFoo = thingy.isFoo

        thingy.isFoo = !oldFoo
    }
}
```

Technically, this works for any data type, not just `Boolean`. In practice, though, you will rarely see Java code use `is...()` for a getter that does not return a `boolean` or `Boolean`.

get() as Square Brackets

Kotlin uses square bracket notation for accessing the contents of things like a `List` or `Map`. In the case of a `List`, the value in the brackets is the 0-based index, while for a `Map`, the value in the brackets is the `Map` key.

Kotlin also extends that notation to any Java class that has appropriate method signatures.

So, if the Java class has `get()` and `set()` methods that work off of an `int` or `Integer` index:

```
public class Something {
    private ArrayList<String> stuff = new ArrayList<>(100);

    public String get(int i) {
        return stuff.get(i);
    }

    public void set(int i, String value) {
        stuff.set(i, value);
    }
}
```

...then Kotlin allows square-bracket notation:

```
class Bar {
    fun doFoo() {
        val thingy = Something()

        thingy[0] = "hello"
        thingy[1] = "world"

        println("${thingy[0]}, ${thingy[1]}")
    }
}
```

In fact, the `get()` and `set()` methods can use any type for their “key”, not just `int`:

```
public class Something {
    private HashMap<String, String> stuff = new HashMap<>();

    public String get(String key) {
        return stuff.get(key);
    }
}
```

```
public void set(String key, String value) {
    stuff.put(key, value);
}
}
```

```
class Bar {
    fun doFoo() {
        val thingy = Something()

        thingy["first key"] = "hello"
        thingy["second key"] = "world"

        println("${thingy["first key"]}, ${thingy["second key"]}")
    }
}
```

It is relatively unlikely that you will have a Java class that conforms to this system, but if it does, you are welcome to use this Kotlin notation to work with it.

Dealing with null

Kotlin has a deep relationship with null:

- Nullable types (String?)
- Safe calls (?.)
- Elvis operator (?:)
- And so on

Most languages do not treat null with the same reverence. In particular, Java does not.

This causes some problems when Kotlin needs to work with Java objects. For example, suppose that we have:

```
public class JavaStuff {
    public String getSomething() {
        return null;
    }
}
```

...and we try using this from Kotlin:

```
val thingy = JavaStuff().something
println(thingy.length)
```

The Kotlin compiler will tend to assume that `getSomething()` is returning `String`, not `String?`, and so it will treat `thingy` as a `String`... and we will crash at runtime with a `NullPointerException`.

If the Java code has annotations that indicate nullability, though, Kotlin will attempt to use them.

```
import androidx.annotation.Nullable;

public class JavaStuff {
    public @Nullable String getSomething() {
        return null;
    }
}
```

Here, we use an Android SDK annotation, `@Nullable`, to indicate that the return value of `getSomething()` might be `null`. Kotlin, in turn, will then treat `getSomething()` as returning `String?` instead of `String`.

Many libraries, such as the Android SDK, use these annotations to help Java and Kotlin developers. However, not all libraries offer them. As a result, be careful when calling Java code from Kotlin, and consider introducing the nullable type yourself. For example, even in the earlier Java example without `@Nullable`, we could write our Kotlin as:

```
val thingy: String? = JavaStuff().something
println(thingy?.length)
```

Here, we are saying that we do not trust `getSomething()` to always return a non-null value, so we force `thingy` to be `String?` for safety's sake.

Java Class Objects

Sometimes in Java, we need to refer to Java Class objects. In Android development, one prominent scenario for this is in creating explicit Intent objects:

```
startActivity(new Intent(this, TheOtherActivity.class));
```

In Java, `.class` as a suffix on the class name returns the `Class` object associated with that class, and we can pass that `Class` object to methods that need one.

In Kotlin, `::class` as a suffix on the class name also returns an object representing the class. However, this is a `KClass` — a Kotlin class object. We need to use `::class.java` to get at the Java `Class` object instead. So this will not compile:

```
startActivity(Intent(this, MainActivity::class))
```

...but this will:

```
startActivity(Intent(this, MainActivity::class.java))
```

There will be cases in Kotlin code where you just use `::class`. That means the function that you are calling takes a `KClass` parameter, probably because that function was written in Kotlin and is designed for use with Kotlin classes.

The SAM Scenario

Every now and then, Java winds up being easier to use from Kotlin than is Kotlin itself.

For example, take this simple Java interface:

```
public interface SomeJavaInterface {  
    void beUseful(String value);  
}
```

Implementations need to override `beUseful()` and... well... be useful.

You can have the equivalent interface in Kotlin, of course:

```
interface SomeKotlinInterface {  
    fun beUseful(value: String)  
}
```

Other than the name, the interfaces seem identical.

However, *using* them from Kotlin is significantly different.

To use the Kotlin interface, we need to create an object that overrides `beUseful()`:

JAVA INTEROPERABILITY

```
val thingy = object : SomeKotlinInterface {  
    override fun beUseful(value: String) {  
        println(value)  
    }  
}
```

To use the Java interface, we could do the same basic thing:

```
val thingy = object : SomeJavaInterface {  
    override fun beUseful(value: String) {  
        println(value)  
    }  
}
```

In practice, though, for Java we can get away with just:

```
val thingy = SomeJavaInterface { println(it) }
```

`SomeJavaInterface` implements the SAM pattern: Single Abstract Method. Kotlin knows how to convert a lambda expression into the implementation of that single abstract method, which in turn allows for the succinct syntax that we see here.

But if you try using that syntax with `SomeKotlinInterface`, you fail with a compile error.

Historically, Kotlin did not support the use of SAM for Kotlin interfaces. The argument boils down to: you should not create `SomeKotlinInterface`. Instead, use a function type, possibly with a typealias:

```
typealias SomeKotlinInterface = (String) -> Unit
```

Now, you can get similar syntax as you get with SAM conversion:

```
val thingy: SomeKotlinInterface = { println(it) }
```

However, Kotlin 1.4 added [functional interfaces](#) which allow for SAM syntax for specially-constructed interfaces.

void and Unit

At this point, though, you may be wondering what `Unit` is.

All functions in Kotlin return something. If you do not specify otherwise, the

function returns `Unit`. This is a singleton: there is exactly one `Unit` in the environment.

You do not see `Unit` mentioned that much because much of the time it can be safely dropped from Kotlin syntax. For example, this function:

```
fun ofMeasure() {  
    println("hello, world!")  
}
```

...is really:

```
fun ofMeasure(): Unit {  
    println("hello, world!")  
  
    return Unit  
}
```

If your function body does not have an explicit return, it returns `Unit`. And, if your function declaration does not have a return type, it returns `Unit`. Kotlin just lets us skip the `Unit` references.

While we can skip `Unit` from an actual function declaration and the return type, a function type cannot skip it. We cannot replace:

```
 typealias SomeKotlinInterface = (String) -> Unit
```

with:

```
 typealias SomeKotlinInterface = (String) ->
```

OR:

```
 typealias SomeKotlinInterface = (String)
```

(the latter compiles but is no longer a function type)

Java methods that “return” `void`, when referenced in Kotlin, really return `Unit`.

Type Mapping

If you have a Java class named `org.yourapp.Restaurant`, and you create an instance of it in Kotlin, you get a `org.yourapp.Restaurant` object.

This may seem obvious.

However, not all types work that way. There is a long list of Java types that get “mapped” to Kotlin native types.

For example:

- Java primitives, like `int`, are turned into instances of Kotlin classes, like `Int`
- “Boxed” Java primitives, like `Integer`, are turned into instances of the corresponding nullable Kotlin class, like `Integer?`
- Certain core Java classes, like `Object` and `String`, get turned into Kotlin equivalents (Any for `Object`, and Kotlin’s own `String` class instead of `java.lang.String`)

The Kotlin documentation has [the full roster of mapped types](#).

Usually, we do not think about this much. However, it may raise some challenges when using reflection or similar techniques that inspect class details at runtime.

Keyword Differences

There is a fair amount of overlap in keywords between Kotlin and Java, such as `class` and `return`. However, there are some Kotlin keywords that are not Java keywords, such as `object` and `when`.

As a result, from time to time, you will see [function names wrapped in backticks](#) — this is to allow you to call, say, a Java `when()` function, despite the fact that `when` is a Kotlin keyword.

Java Calling Kotlin

If you are embarking on a new Kotlin-focused project, you might think that your only concern is having Kotlin code call Java code in libraries.

However, frequently, Java code needs to call other Java code that we supply:

- Via callbacks
- Via subclasses (e.g., calls to methods that we override)

We need to do those things in Kotlin too, which means that some Java code is going

to need to call to Kotlin code that we write.

For example, in Android development, `Activity` and `AppCompatActivity` are Java classes, but a Kotlin-focused project will create subclasses in Kotlin. As a result:

- You will need to override Java methods like `onCreate()`, creating equivalent Kotlin functions, and Java code in the framework or AndroidX libraries will wind up calling your overridden methods
- You may create event listeners, for things like button clicks, which will result in Java classes like `Button` calling your Kotlin listeners

Normally, this all works without much issue. However, from time to time, we do have to worry a bit about how Java calls our Kotlin code.

Public Properties

Kotlin properties look like Java fields. In reality, a Kotlin property is a combination of:

- A getter function
- A setter function (for mutable `var` properties)
- A “backing” field that actually stores the data

In Kotlin, we frequently ignore this distinction. In Java, we cannot ignore it, as a public property is *not* the same as a public Java field. We have no direct access to the field from Java, but we can work with the getter (and, where available, the setter).

So, if we have this Kotlin class:

```
class KotlinStuff {
    val immutableStuff = "foo"
    var mutableStuff = "bar"
}
```

...then we can use the getters for both properties and the setter for `mutableStuff` from Java:

```
public class JavaStuff {
    public void useKotlinStuff(KotlinStuff stuff) {
        String immutable = stuff.getImmutableStuff();
        String mutable = stuff.getMutableStuff();
    }
}
```



```
stuff.setMutableStuff("and now for something completely different");
}
}
```

Kotlin automatically creates the getter and setter using standard JavaBean naming rules, the same ones that it handles [in the Kotlin-calling-Java direction](#).

By default, these will be `get...()` and `set...()`, where `...` is replaced by the name of the property, with the initial letter capitalized (`immutableStuff` results in `getImmutableStuff()`). This is true even for Boolean properties — Kotlin does not switch to `is...()` syntax automatically.

However, if the property name itself begins with `is`, then Kotlin will have the getter function use the same name as the property, with a setter function that replaces `is` with `set`. So, a Kotlin `isMutable` property:

```
class KotlinStuff {
    var isMutable = true
}
```

...results in `isMutable()` and `setMutable()` Java methods for us to invoke on the Java side:

```
public class JavaStuff {
    public void useKotlinStuff(KotlinStuff stuff) {
        if (stuff.isMutable()) {
            stuff.setMutable(false);
        }
    }
}
```

Note that class modifiers like `data` or `sealed` have no impact on these naming rules, so this `KotlinStuff` variant works the same as the previous one from Java's standpoint:

```
data class KotlinStuff(var isMutable: Boolean = true)
```

Other Top-Level Declarations

As we have seen throughout this book, Kotlin supports top-level functions: functions that reside outside of any Kotlin class:

JAVA INTEROPERABILITY

```
fun doSomething() {  
    println("something!")  
}
```

Since Java does not have its own top-level function, it needs a little help to call those functions.

From Java's standpoint, the top-level function is a static method on a class whose name is:

- the name of the Kotlin source file containing the top-level function...
- ...with a Kt suffix added to it

If the `doSomething()` function shown in the above snippet were in a `KotlinStuff.kt` source file, then Java can refer to it as a static method on `KotlinStuffKt`:

```
public class JavaStuff {  
    public void doSomethingElse() {  
        KotlinStuffKt.doSomething();  
    }  
}
```

Top-level properties are also accessible from Java. However, like class properties, Java will use generated getter and setter methods by default. Those appear as static methods on the Java class generated from the class name, as with top-level functions.

So, if `KotlinStuff.kt` has:

```
val GLOBAL_VARIABLE = "it's a small world"
```

...then Java can obtain the value via:

```
public class JavaStuff {  
    public void doSomethingGlobal() {  
        String value = KotlinStuffKt.getGLOBAL_VARIABLE();  
    }  
}
```

One exception is if `const` is used:

```
const val GLOBAL_VARIABLE = "it's a small world"
```

Then, the value is accessible in Java as a static field:

```
public class JavaStuff {
    public void doSomethingGlobal() {
        String value = KotlinStuffKt.GLOBAL_VARIABLE;
    }
}
```

This is one advantage of using `const val` instead of just `val`: you have a more natural way of referencing the constant from Java.

Adjustments via Annotations

Sometimes, you are going to need to have Kotlin change its approach slightly to be more Java-friendly. Through a series of annotations, you can alter how Kotlin code can be used from Java, without interfering with how that same Kotlin code can be used from Kotlin itself.

@file:JvmName()

As we just saw, top-level Kotlin functions can be called as `static` methods from Java, using a synthetic Java class based on the Kotlin filename. Sometimes, though, that filename is not particularly convenient. You may have chosen to put that Kotlin code in `ASourceFileWithAVeryLongName.kt`, forcing you to call its top-level functions on `ASourceFileWithAVeryLongNameKt`. Or, perhaps the filename is short, but it does not make sense as a Java class name. Or, perhaps you just do not like the `Kt` suffix on the end.

To control this, put a `@file:JvmName()` annotation as the first line of the source file containing the top-level functions that you want to use from Java. You supply the annotation with the name that you want Kotlin to use for the synthetic Java class:

```
@file:JvmName("CoolStuff")
val GLOBAL_VARIABLE = "it's a small world"
```

...then Java can obtain the value by using your supplied class name:

```
public class JavaStuff {
    public void doSomethingGlobal() {
        String value = CoolStuff.getGLOBAL_VARIABLE();
    }
}
```

@file:JvmMultifileClass

If you use the same `@file:JvmName()` annotation (with the same name) in 2+ Kotlin source files, you will get a build error. The assumption is that you made a copy-and-paste error, forgetting to change the name.

However, Kotlin actually supports having more than one Kotlin file contribute to the same Java facade class for things like top-level functions, object references, and so on. You just have to opt into it by also having `@file:JvmMultifileClass` as an annotation, typically immediately after the `@file:JvmName()` annotation:

```
@file:JvmName("CoolStuff")
@file:JvmMultifileClass

// this is in one Kotlin file

val GLOBAL_VARIABLE = "it's a small world"
```

```
@file:JvmName("CoolStuff")
@file:JvmMultifileClass

// this is in another Kotlin file

val OTHER_VARIABLE = "...but we can make it bigger!"
```

Your Java code can reference both `CoolStuff.GLOBAL_VARIABLE` and `CoolStuff.OTHER_VARIABLE`, even though those two variables are defined in separate Kotlin source files.

@JvmStatic

Whether you use `@file:JvmName()` or not, Java can call top-level Kotlin functions.

When it comes to functions on a companion object, things get a bit more complicated.

For example, suppose that you have a Kotlin class that has a companion object with some sort of factory function for creating instances:

JAVA INTEROPERABILITY

```
class KotlinThingy {
    companion object {
        fun gimme() = KotlinThingy()
    }
}
```

Java can call this by referring to a Companion synthetic nested class:

```
public class JavaStuff {
    public void doSomethingGlobal() {
        KotlinThingy thingy = KotlinThingy.Companion.gimme();
    }
}
```

There is nothing wrong with this. However, if you want, you can add `@JvmStatic` to the companion object function:

```
class KotlinThingy {
    companion object {
        @JvmStatic
        fun gimme() = KotlinThingy()
    }
}
```

The `KotlinThingy.Companion.gimme()` syntax will still work in Java, but now you also can call a `gimme()` static method on the Kotlin class, much like how you use the companion object functions in Kotlin itself:

```
public class JavaStuff {
    public void doSomethingGlobal() {
        KotlinThingy thingy = KotlinThingy.gimme();
    }
}
```

Another place where `@JvmStatic` can be useful is with named objects:

```
object KotlinSingleton {
    fun heyNow() {
        println("What what?")
    }
}
```

Java can call `heyNow()`, but it needs to refer to an INSTANCE of the object:

```
public class JavaStuff {
    public void doSomethingGlobal() {
        KotlinSingleton.INSTANCE.heyNow();
    }
}
```

If, instead, the Kotlin code uses `@JvmStatic`:

```
object KotlinSingleton {
    @JvmStatic
    fun heyNow() {
        println("What what?")
    }
}
```

...then the Java code can skip the `INSTANCE` and call `heyNow()` directly on the Kotlin object:

```
public class JavaStuff {
    public void doSomethingGlobal() {
        KotlinSingleton.heyNow();
    }
}
```

@Throws

Kotlin does not have checked exceptions the way Java does — you are never *required* by the Kotlin compiler to wrap something in a try/catch construct. All JVM-defined exceptions are treated as runtime exceptions by Kotlin. And we have no `throws` keyword in Kotlin to advertise that a certain function will throw an exception.

For exceptions that extend Java's `RuntimeException`, this is fine.

We start to run into problems when Kotlin code might throw some other sort of exception (e.g., `IOException`) *and* we want that exception to be caught by Java code calling the Kotlin code. The Java compiler expects all checked exceptions to be reported by a `throws` keyword somewhere, and it will refuse to compile code that tries to catch an exception that is not a `RuntimeException` *and* was not reported by `throws`.

So, if we have some Kotlin code that might throw a checked exception:

JAVA INTEROPERABILITY

```
class KotlinThingy {
    fun loadStuff() {
        // TODO work that might throw an FileNotFoundException, which we simulate by...

        throw FileNotFoundException()
    }
}
```

...and we try to catch it in Java:

```
public class JavaStuff {
    public void useTheLoadedStuff() {
        try {
            KotlinThingy thingy = new KotlinThingy();

            thingy.loadStuff();
        }
        catch (FileNotFoundException ex) {

        }
    }
}
```

...we get a Java compile error, complaining that `loadStuff()` does not throw `FileNotFoundException`.

The workaround for this is to annotate the Kotlin function with `@Throws`, identifying the exception(s) that the function may throw:

```
class KotlinThingy {
    @Throws(FileNotFoundException::class)
    fun loadStuff() {
        // TODO work that might throw an FileNotFoundException, which we simulate by...

        throw FileNotFoundException()
    }
}
```

This does not change the syntax used by Java to call the function, but it allows the catch to work.

If the function throws more than one checked exception, supply an array of exception classes to the `exceptionClasses` property:

```
class KotlinThingy {
    @Throws(exceptionClasses = [FileNotFoundException::class,
CertificateException::class])
    fun loadStuff() {
        // TODO work that might throw either of those exceptions
    }
}
```

@JvmField

As was noted earlier in the book, a Kotlin property is made up of three pieces:

- A field
- A getter method
- A setter method

Frequently, we ignore the field in Kotlin, as our references to the property automatically use the getter and setter. Similarly, by default, Java cannot access the field — only the Kotlin getter and setter are exposed to Java.

However, there may be rare occasions where you have Java code that needs to access a Kotlin field directly. In those cases, the `@JvmField` annotation on the Kotlin property will make the field available, with whatever visibility is defined on the property (public by default, but could be protected, etc.).

@JvmOverloads

Kotlin supports default parameter values on constructors and functions. Java, through Java 8, does not. By default, what Kotlin exposes to Java will be a function that requires all parameters, meaning that the default values will be ignored.

For example, if you have this Kotlin code:

```
class KotlinThingy {
    fun deeeeeeeefault(foo: String, bar: Int = 1, goo: Float = 2.3f) {
        // TODO something
    }
}
```

...you might think that this Java could would work:


```
public class JavaStuff {
    public void iCanHazDefaults() {
        KotlinThingy thingy = new KotlinThingy();

        thingy.deeeeeeeeeefault("non-default");
    }
}
```

However, the Java compiler will complain that you do not have values for the second and third parameters. Even though Kotlin callers do not require them, Java callers do.

There are two exceptions to this rule.

One is with constructors. If your class' constructor has default values for *all* constructor parameters, then a secondary zero-parameter constructor is also available for Java. So, if we revise the above Kotlin code to:

```
class KotlinThingy(parameters: String = "", are: Int = 4, fundamental: Boolean = true) {
    fun deeeeeeeeeefault(foo: String, bar: Int = 1, goo: Float = 2.3f) {
        // TODO something
    }
}
```

...then the Java code will still be able to create a `KotlinThingy` using `new KotlinThingy()`, even though the Kotlin constructor uses default parameter values.

The other exception to this rule is if you apply `@JvmOverloads` to the constructor or function. This causes the Kotlin compiler to generate additional versions of the constructor or function, progressively applying default parameters and therefore not requiring Java to supply them.

We can apply that to `KotlinThingy` both for the constructor and the function:

```
class KotlinThingy @JvmOverloads constructor(parameters: String, are: Int = 4, fundamental: Boolean = true) {
    @JvmOverloads fun deeeeeeeeeefault(foo: String, bar: Int = 1, goo: Float = 2.3f) {
        // TODO something
    }
}
```

This version of the constructor only has the latter two parameters defaulted; the first one is required.

In Java, we now have access to three KotlinThingy constructors:

- KotlinThingy(String, int, boolean)
- KotlinThingy(String, int)
- KotlinThingy(String)

Similarly, we have access to three versions of the function:

- deeeeeeeeeefault(String, int, float)
- deeeeeeeeeefault(String, int)
- deeeeeeeeeefault(String)

As a result, Java code like this will compile cleanly:

```
public class JavaStuff {
    public void iCanHazDefaults() {
        KotlinThingy thingy = new KotlinThingy("whatever");

        thingy.deeeeeeeeeefault("non-default");
    }
}
```

Prefixes on Other Annotations

Java has its own annotations, and sometimes you will need to apply an annotation defined in Java to some Kotlin code.

Normally, this works fine.

However, sometimes you need to apply an annotation to something *specific* that Kotlin hides behind other layers.

The biggest culprit here are Kotlin properties. These map to a field, a getter, and a setter in Java. If you need to put the annotation on just one of those things, the annotation alone will not work. Instead, you have to add a prefix to the annotation itself:

- field: to apply the annotation to the “backing field” of the property (@field:YourAnnotation)
- get: to apply the annotation to the getter (@get:YourAnnotation)
- set: to apply the annotation to the setter (@set:YourAnnotation)

Scenario: JUnit4 Rules

For example, if you use JUnit4 for testing, you might want to use some [JUnit4 rules](#). In Java, you apply the `@Rule` annotation to the field that holds the rule:

```
public class YourTest {
    @Rule
    public final TemporaryFolder temp = new TemporaryFolder();

    // TODO rest of testing
}
```

Here, we use the stock JUnit 4 `TemporaryFolder` rule, to generate a temporary folder suitable for our testing.

The direct equivalent in Kotlin is to put the annotation on a property:

```
class YourTest {
    @Rule
    val temp = TemporaryFolder()

    // TODO rest of testing
}
```

However, this confuses the annotation processor. Instead, you can use `@get:Rule` to apply the annotation to the getter:

```
class YourTest {
    @get:Rule
    val temp = TemporaryFolder()

    // TODO rest of testing
}
```

Scenario: Dagger 2 Qualifiers

You can use custom annotations with Dagger 2 to help qualify particular uses of objects.

For example, you might define a `@SpecialCase` custom annotation:

```
@Qualifier
@Retention(AnnotationRetention.RUNTIME)
annotation class SpecialCase
```

...then use that for some object defined in a module:

```
@Module
class YourModule {
    @Provides
    @Singleton
    @SpecialCase
    fun heyThisIsSpecial() = Whatever()
}
```

You then might want to @Inject that @SpecialCase object as a property somewhere. To make that work successfully, you may need to use @field:SpecialCase to tell Kotlin to apply the @SpecialCase annotation to the backing field of the property:

```
abstract class SomethingOrAnother {
    @Inject @field:SpecialCase lateinit var special: Whatever

    // TODO rest of this class, including an inject() call on a Dagger component
}
```

Decompiling Kotlin to Java

Sometimes, to debug Java/Kotlin interoperability issues, it helps to see the equivalent Java code for a particular Kotlin source file.

IntelliJ IDEA and Android Studio both offer this capability. From the Tools > Kotlin menu, choose “Show Kotlin Bytecode”. This will open up the “Kotlin Bytecode” view and will show you the JVM bytecode associated with the current Kotlin source file. In that view, the “Decompile” button will attempt to decompile that bytecode into Java source code, opening up the result in another editor tab.

This works but is not without its flaws:

- Decompiled bytecode can be more difficult to read, as local variables tend to use generated names (e.g., var2) rather than human-readable names
- The decompilation process can be very slow, depending on both the direct size of the Kotlin source file and what that source uses that needs to be converted
- On Android Studio in particular, [it may take a bit of work to get the decompiler to run](#)

Changes in Newer Kotlin Versions

Kotlin keeps changing.

Generally newer versions of Kotlin are backwards-compatible with older ones. So, when we switch to a newer version of Kotlin (say, in `build.gradle` of an Android app project), most of our existing code should still work.

However, newer Kotlin versions offer new features that we can use in our development.

In this chapter, we will look at changes in the two newest Kotlin versions: 1.3 and 1.4.

Version 1.3

In November 2018, JetBrains released Kotlin 1.3. This was a relatively small update over Kotlin 1.2, but it did add a few new features of note. However, even today, some of those features are still marked as pre-release — not everything in a stable Kotlin version is stable.

With that in mind, here are some of [the changes introduced in Kotlin 1.3](#).

`when()` and Subject Variables

As we saw earlier in the book, you can use an expression in `when()` to provide the value for comparison in each of the branches:

```
val thingy = "foo"

when (thingy) {
    "foo" -> println("something")
}
```

CHANGES IN NEWER KOTLIN VERSIONS

```
"bar" -> println("something else")
else -> println("and now for something completely different")
}
```

(from ["when Works with Any Type" in the Klassbook](#))

Starting with Kotlin 1.3, you can make that expression also be stored in a local variable:

```
fun thingyProvider() = "like, whatever"

fun main() {
    when (val thingy = thingyProvider()) {
        "foo" -> println("something")
        "bar" -> println("something else")
        else -> println("we received: $thingy")
    }
}
```

(from ["Capturing Values in when" in the Klassbook](#))

The variable's scope is the `when()` itself, so you can reference it from the branches but not anywhere else.

This is particularly useful for `else` branches, such as for logging unexpected values.

JVM Interop for Interfaces

If you have a companion object for an interface, you have improved interoperability options with Java:

- You can mark properties in the companion object with `@JvmField`
- You can mark functions in the companion object with `@JvmStatic`

In both cases, from Java's standpoint, these become static members of the interface.

So, suppose we had:

```
interface Something {
    companion object {
        @JvmField
        val tag: String = "Something!"

        @JvmStatic
```

```
        fun printMe() {
            println(tag)
        }
    }

    // TODO add rest of interface
}
```

From Java code, we can reference `Something.tag` and `Something.printMe()`.

Inline Classes

Later in the book, we will explore [inline functions](#). Simply put, a function marked with `inline` works like a regular function, but the compiler “inlines” its implementation at each place it is used.

Kotlin 1.3 added a related capability, called inline classes. This feature is still deemed to be in alpha state, but it offers some interesting possibilities. We will explore inline classes more [later in the book](#).

Contracts

Earlier in the book, we covered [smartcasts](#), where Kotlin will allow syntax that on the surface seems impossible because it infers that it *is* possible under the circumstances.

```
open class Animal

class Frog : Animal() {
    fun hop() = println("Hop!")
}

class Axolotl : Animal() {
    fun swim() = println("Swish!")
}

fun main() {
    val critter: Animal = Frog()

    when (critter) {
        is Frog -> critter.hop()
        is Axolotl -> critter.swim()
    }
}
```


(from ["Smart Casts" in the Klassbook](#))

Here, we can `hop()` and `swim()` on `critter` because the compiler knows more about the type of `critter` given the rules encoded in the `when` branches.

However, prior to Kotlin 1.3, there were only a few places where smartcasts could help. Kotlin 1.3 adds a system of “contracts” that allows this sort of behavior in more places... including in code written by you (though this part is experimental).

Tactically, it means that if you have Kotlin 1.2 or older code, when you move to Kotlin 1.3, you may find that your IDE or linter will report a bunch of places where you can remove manual casts, unnecessary `null` checks, and the like. These will reflect the contracts that were added to the Kotlin standard library.

Long-term, “custom contracts”, written by ordinary developers, will be an option. As noted above, this is considered experimental. [This document](#) outlines the approach.

Unsigned Integer Types

Kotlin 1.3 added [support for unsigned numeric types, such as `UInt` and `ULong`](#).

Note, though, that these are still considered to be in beta form. Unless you have a specific reason to use them (e.g., simplify interoperability with some JNI code in Android), it is probably better to leave them alone for now.

Version 1.4

August 2020 brought us Kotlin 1.4, which has [a longer roster of developer-facing changes](#), though they are of mixed importance. This section examines some that may be encountered more often by ordinary Kotlin developers.

Exception Changes

One change that might cause you to need adjustments to your code is in how Kotlin raises null pointer exceptions in Kotlin/JVM, including in Android apps.

Formerly, a null pointer resulted in any number of possible exceptions, including:

- `IllegalArgumentException`
- `IllegalStateException`
- `KotlinNullPointerException`

- `TypeCastException`

...in addition to Java's `NullPointerException`.

In Kotlin 1.4, Kotlin now reliably generates a `NullPointerException` in all cases.

Probably you do not have code that depends on the particular type of one of these exceptions. That is because usually we do not explicitly worry about this exception compared to others. But, if you had a `try/catch` that depended upon a null pointer being raised as one of the former exception types, that code will need to be adjusted.

Trailing Commas

We have a lot of comma-delimited things in Kotlin, particularly all of our parameter lists:

```
val spices = mutableListOf("Ginger", "Posh", "Scary", "Sporty", "Baby")
```

Now, we can have trailing commas, without causing a syntax error. This is handy when the entries are all on individual lines, particularly for copying and pasting:

```
val spices = mutableListOf(  
    "Ginger",  
    "Posh",  
    "Scary",  
    "Sporty",  
    "Baby",  
    "Pumpkin",  
)
```

The trailing comma is ignored by the language parser.

Mixed Named and Positional Parameters

It used to be that named parameters could only come at the end of a call:

```
doSomething(2, 4, 6, 8, whoDoWeAppreciate = "everyone!")
```

The idea was that named parameters mostly would be used for optional parameters, where the function had declared default values.

However, a secondary use of named parameters is simply for documentation. For

that, Kotlin 1.4 now supports named parameters in arbitrary spots:

```
fun doSomething(  
    colors: Int,  
    sides: Int,  
    quantity: Int,  
    theseParametersAreNotSerious: Int,  
    whoDoWeAppreciate: String = ""  
)  
  
doSomething(2, sides = 4, quantity = 6, 8, whoDoWeAppreciate = "everyone!")
```

Basically, up through the last positional parameter, the names are purely documentation — the position of the parameter is what matters.

Functional Interfaces

Earlier in the book, we saw [SAM](#), the Single Abstract Method pattern. It takes code like this:

```
val thingy = object : SomeJavaInterface {  
    override fun beUseful(value: String) {  
        println(value)  
    }  
}
```

...and shrinks it down to code like this:

```
val thingy = SomeJavaInterface { println(it) }
```

However, this was limited to interfaces that had a single abstract method (e.g., `beUseful()`). And, this was limited to *Java* interfaces — Kotlin interfaces with a single abstract method did not support the SAM usage pattern.

Kotlin 1.4 relaxes that a bit. You can use SAM syntax with a Kotlin interface... if that interface is fun:

```
fun interface Comparitizer<T> {  
    fun valid(item: T): Boolean  
}  
  
fun <T> firstItemOrNull(items: Collection<T>, comparitizer: Comparitizer<T>): T? {  
    items.forEach {  
        if (comparitizer.valid(it)) return it  
    }  
}
```

CHANGES IN NEWER KOTLIN VERSIONS

```
    return null
}

data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    val leetEvent = firstOrNull(events, Comparitizer { it.id == 1337 })

    println(leetEvent)
}
```

(from ["Functional Interfaces" in the Klassbook](#))

Marking the interface itself with the `fun` keyword makes it a “functional interface”, one that supports SAM syntax. However, a functional interface imposes one key limit: there can only be one abstract function (a.k.a., single abstract method). The `fun` interface can have as many concrete functions as are needed, though.

Delegated Properties Modifications

Kotlin 1.4 added a new feature to property delegates: the ability delegate a property to another property. We will explore that [later in the book](#).

Explicit API Mode

For those of you considering publishing a Kotlin library, Kotlin 1.4 adds what JetBrains calls “explicit API mode”. You can opt into this for a library module or project. It adds additional compiler checks to help ensure that your public API is what you think it is. Quoting [the proposal that led to this](#):

Compilation in such mode differs from the default mode in the following aspects:

- Compiler requires you to specify explicit visibility for a declaration when leaving default visibility would result in exposing that declaration to the public API.
- Compiler requires you to specify the explicit type of property/function when it is exposed to the public/published API.
- Compiler requires you to explicitly propagate experimental status for functions which contain experimental types in the signature.
- Compiler warns you when declaration exposed to public API does not have a KDoc.

CHANGES IN NEWER KOTLIN VERSIONS

[The documentation](#) has additional details, such as how to enable this mode.

Kotlin, WTF?

Custom Accessors

Sometimes, you will see a `get()` or perhaps a `set()` associated with a property. Those are custom accessors, replacing the stock ones that Kotlin would generate for you.

Defining Custom Accessors

Each Kotlin property is made up of a field, a getter function, and a setter function. By default, the getter and setter are code-generated for you, but you can override those with custom implementations if desired.

For example, here is a property with a custom getter function:

```
val stuff = mapOf("something" to "This is the 'something' value")
val something: String?
    get() = stuff["something"]

fun main() {
    println(something)
}
```

(from ["Custom Getter Functions" in the Klassbook](#))

The custom getter goes on the line after the property declaration. It is declared as a `get()` function and needs to return an object matching the property's data type. Otherwise, though, the actual function implementation is up to you. In this case, we declare that the `something` value really comes from the `stuff` Map, rather than from the field that ordinarily would be the value of `something`.

Not surprisingly, you can override the setter function too. That requires a `var` property, as a `val` property has no setter. The setter appears after the property,

CUSTOM ACCESSORS

alongside the getter (if there is one). The setter is a `set(value)` function, where `value` is the value that is being set. So in this example, we turn around and put that value into the `stuff` using our `something` key:

```
val stuff = mutableMapOf<String, String?>("something" to "This is the 'something' value")
var something: String?
    get() = stuff["something"]
    set(value) { stuff["something"] = value }

fun main() {
    something = "This is different"
    println(something)
}
```

(from "[Custom Setter Functions](#)" in the *Klassbook*)

Note that users of the property are not affected by these changes. You use the same syntax for working with the property as before.

Referencing the Field

If you declare a normal simple property, Kotlin will create the corresponding field into which the data gets stored. The generated getter returns the value of the field, while the setter replaces the value in the field.

If you declare custom accessors, by default Kotlin will not generate a corresponding field... unless you choose to reference it from the custom accessor functions via the `field` name. Then, Kotlin creates that field and `field` serves as a reference to it. If your accessors reference `field`, though, your property needs an initializer, to supply the initial value for `field`.

So, this is basically what the default Kotlin code generation would look like if you did it manually:

```
var something: String = "This is the 'something' value"
    get() = field
    set(value) { field = value }

fun main() {
    something = "This is different"
    println(something)
}
```

(from "[Custom Accessors and the Field](#)" in the *Klassbook*)

Inline Getters

For a `val` property, if your custom getter clearly indicates the data type of the property, you can skip putting the data type on the property itself. Instead, put the custom getter function immediately after the property name:

```
val stuff = mapOf("something" to "This is the 'something' value")
val something get() = stuff["something"]

fun main() {
    println(something)
}
```

(from "[Inline Custom Getter Functions](#)" in the *Klassbook*)

Configuration Without Customization

Sometimes, you do not need to fully customize the accessor, but simply configure it slightly.

For example, you might have a `var` property in a class, because you need to assign a new value to it after initialization. However, modifying that property should be limited to instances of the class itself, whereas reading that property might be available to all classes. In other words, you want a semi-public property: public getter and private setter.

That is just a matter of having a `private set` declaration, without a function body:

```
class Whatever {
    var something: String = "This is the 'something' value"
    private set

    // TODO
}
```

Instances of `Whatever` can modify the `something` property, while everything else is limited to reading that property.

Anti-Patterns

It is tempting to put data conversion or normalization in a custom setter. However, this means that the getter will not return what was last set in the setter:

CUSTOM ACCESSORS

```
var something: String = "This is the 'something' value"
    get() = field
    set(value) { field = value.toUpperCase() }

fun main() {
    println(something)
    something = "This is different"
    println(something)
}
```

(from "[Custom Accessor Parallelism](#)" in the *Klassbook*)

Running this gives us:

```
This is the 'something' value
THIS IS DIFFERENT
```

Not only does the value returned by the second something reference not match what we set on it, but the initializer bypasses the setter, possibly resulting in “invalid” data.

Also, be very careful when making a custom accessor be expensive, such as being slow. Developers will assume that accessors are cheap and may not realize the cost of calling your custom ones.

Alternative: Delegates

If you expect to repeat the same sort of custom getters and setters, you might consider implementing [a delegate](#) instead. A delegate will allow you to encapsulate the common logic better.

Extension Properties

Not only can you create extension functions, but you can create extension *properties* as well! The resulting syntax resembles a combination of extension functions and custom property accessors.

Recapping Extension Functions

We saw extension functions [earlier in the book](#). You can use them to add functions to existing classes, including classes that you did not create:

```
private val EMAIL_REGEX = Regex("[a-zA-Z0-9_.-]+@[a-zA-Z0-9-]+[.]+[a-zA-Z0-9-]+(\\s)*")
fun String.isValidEmail() = EMAIL_REGEX.matches(this)

fun main() {
    println("martians-so-do-not-exist@commonsware.com".isValidEmail())
    println("this is not an email address".isValidEmail())
}
```

(from "[Extension Functions](#)" in the *Klassbook*)

From a declaration standpoint, rather than a simple function name, the declaration uses `ClassName.functionName()`, where `ClassName` is the class that you are extending with the extension function and `functionName` is the name of the function that you are adding.

As noted in that earlier chapter, though, extension functions cannot add properties to a class.

Recapping Property Custom Accessors

As we saw in [the preceding chapter](#), you can override the getter and/or setter for a

EXTENSION PROPERTIES

regular property:

```
val stuff = mapOf("something" to "This is the 'something' value")
val something: String?
    get() = stuff["something"]

fun main() {
    println(something)
}
```

(from ["Custom Getter Functions" in the Klassbook](#))

```
val stuff = mutableMapOf<String, String?>("something" to "This is the 'something' value")
var something: String?
    get() = stuff["something"]
    set(value) { stuff["something"] = value }

fun main() {
    something = "This is different"
    println(something)
}
```

(from ["Custom Setter Functions" in the Klassbook](#))

These allow you to tailor the use of the property, to the point where you might not actually use the backing field of the property at all.

Extension Properties = A Mashup

An extension property looks like a regular property with custom accessors, except that it is declared on an existing class, the way an extension function is:

```
val <T> Map<String, T>.something: T?
    get() = this["something"]

val stuff = mapOf("something" to "This is the 'something' value")

fun main() {
    println(stuff.something)
}
```

(from ["Extension val Property" in the Klassbook](#))

This is the same basic scenario as the custom getter example from above, except that we move it to be an extension property on `Map<String, T>` for some generic type `T`. If you ignore the generics, the syntax is pretty much the same as it is for extension functions: `ClassName.propertyName` to declare that this property is to be added to

EXTENSION PROPERTIES

ClassName.

With an extension `val` property, we have to provide a custom getter. The value of the property has to come from somewhere, and it is up to us to define what that “somewhere” is. In this case, we just obtain the value of the “something” entry in the Map, if there is one.

If you use `var` instead of `val`, you will need to provide both the custom getter and the custom setter:

```
var <T> MutableMap<String, T>.something: T
    get() = this["something"] ?: throw IllegalArgumentException("we do not have something")
    set(value) { this["something"] = value }

fun main() {
    val stuff = mutableMapOf<String, String?>("something" to "This is the 'something' value")

    stuff.something = "This is different"
    println(stuff.something)
}
```

(from "[Extension var Property](#)" in the *Klassbook*)

You might wonder why the custom getter works. `[]` syntax on a Map returns a nullable type (`T?`), and our custom getter needs to return `T`. So, using the Elvis operator, we throw an exception if `[]` returns `null`. The reason why the compiler then realizes that our custom getter returns a `T` has to do with what `throw` evaluates to, as we will see [later in the book](#).

Extension properties, therefore, are little more than extension functions, just ones that provide users with property syntax instead of function syntax.

Escaping Keywords

In other words, why does `when()` become ``when`()`?

The Scenario: Mockito

[Mockito](#) is a popular mocking framework for Java and Kotlin. You can use it in unit tests (and Android instrumentation tests) to create mock implementations of objects, where you can have your tests control how those objects respond to various function calls.

A core function for that is `Mockito.when()`, used to teach a Mockito mock how to respond to a particular Java method call, such as in this Java snippet:

```
Mockito.when(thisThing.isCalledWithThis()).thenReturn("something");
```

...or, if you use a static import of `when()`:

```
when(thisThing.isCalledWithThat()).thenReturn("something else");
```

The Problem: Keywords

In Java, `when` is not a keyword.

In Kotlin, `when` is a keyword. And, like many programming languages, keywords are reserved symbols. You cannot use those symbols for your own function names, properties, and the like.

Every now and then, you will encounter a Java library, like Mockito, that uses Kotlin keywords for method names. Since the library is already compiled, the library itself

is fine. And if the collision on the Kotlin keyword is purely part of the internal implementation of the library, everything is fine. But, if the collision is in the public API of the library, developers using Kotlin would be unable to call that API.

The Solution: Backticks

`when()` is a reference to the Kotlin construct.

By contrast, ``when`()` (so, `when` with backticks around it) is not. The backticks indicate that this is something else that just happens to be named `when`.

So, to use Mockito's `when()` method, you can add the backticks to the `import` of that static method:

```
import org.mockito.Mockito.`when`
```

Then, you can use backticks on the call:

```
`when`(thisThing.isCalledWithThat()).thenReturn("something else");
```

If you find that syntax to be ugly, and if there is another likely name to use for the function, you could limit the backticks to just the `import` statement, plus use as to [rename the import](#). This would allow you to avoid the backticks on every usage of the collision.

Escaped Method Names

```
fun `is this syntax really necessary?`() {  
    println("Well, it has its uses...")  
}  
  
`is this syntax really necessary?`()
```

The Scenario: JUnit Tests

Most of the time, function names are used purely within our code. Occasionally, though, function names get exposed outside of the code itself.

One such case is with JUnit, particularly the JUnit4 used for years in Android app development. JUnit is a popular unit testing harness and framework for Java and JVM languages. JUnit — and tools that use it, like IDEs and continuous integration (CI) servers — have “test runners” that execute designated test functions. By default, the output of the tests includes the names of the test classes and the test functions.

The Problem: You Are Tired of CamelCase

Convention says that the test function names should indicate what the test function actually tests. As a result, they tend to be relatively verbose, even more so than Java normally gets. A test function with 6+ words chained together is far from unusual.

Java has strict rules on method names. That, coupled with standard Java conventions, means that test functions wind up beingChainedTogetherUsingCamelCase().

This is quick, easy, but not particularly readable. That is especially true if non-

programmers are going to be reading test output: dedicated QA staff, managers, etc.

The Solution: Backticks

Kotlin has naming rules for functions that are very similar to Java's naming rules for methods, with one key difference: in Kotlin, you can use backticks around a function name. Within the backticks, just about anything is valid for the function name itself. To call such a function, you use the same function name, including the backticks

So, going back to the sample at the outset, this function declaration and invocation is perfectly legal:

```
fun `is this syntax really necessary?`() {  
    println("Well, it has its uses...")  
}  
  
`is this syntax really necessary?`()
```

For ordinary code, this would get tedious, particularly if you needed to call the function a lot. In the case of something like JUnit, though, you usually do not write code that calls the test functions — the test runner calls them via reflection. So the function declaration is the only place where the backticks get used. And, given the backticks, you can name the function whatever you want — including spaces and punctuation — to make it easier to read in test output.

Note, though, that support for such function names may vary by environment. Java cannot handle such function names, so any place where Java might need to work with the actual name runs the risk of encountering problems. For example, in Android app development, escaped function names with backticks work fine for unit tests but not for instrumented tests.

Property Delegates

In other words, let's find out more about how to be lazy!

A Refresher on lazy

Back in [the chapter on properties](#), we saw by lazy syntax:

```
class Foo(val rawLabel: String) {
    var count = 0
    val label: String by lazy { println("initializing via lazy!"); rawLabel.toUpperCase() }

    fun something() {
        count += 1
        println("$label: something() was called $count times")
    }
}

fun main() {
    val foo = Foo("foo")

    foo.something()
    foo.something()
    foo.something()

    println("the final count was ${foo.count}")
}
```

(from "[Lazy Properties](#)" in the *Klassbook*)

Here, label is computed lazily. If the code were:

```
val label = rawLabel.toUpperCase()
```

...then label would be populated immediately. Instead, by lazy will evaluate the supplied lambda expression when label is first accessed.

In this particular example, there is no practical difference, since `rawLabel` is a `val` property and will not change during the lifetime of `label`, and we are always accessing `label`. However, `by lazy` is useful if:

- The initialization might not be needed, and
- The initialization is a bit expensive

What's Really Going On

`by lazy` is really two things: `by` and `lazy`. `by` sets up a property delegate, while `lazy` is a particular delegate with particular behavior.

`by`

A Kotlin property is made up of a field, a getter, and a setter. Frequently, the getter and setter are code-generated for us.

There are basically two ways to replace those code-generated accessors:

1. Override them [with custom implementations](#) directly
2. Redirect them to a property delegate

The expression to the right of `by` needs to evaluate to a property delegate. That delegate will have `getValue()` and `setValue()` functions, and those functions will replace the stock getter and setter, respectively. So, when you try to read the property value, the delegate's `getValue()` function is called, and when you try to write the property value, the delegate's `setValue()` function is called. Whatever those functions do represent what the property does.

`lazy`

`lazy()` is a Kotlin-supplied top-level function that returns a `Lazy` property delegate.

The `lazy()` function takes an “initializer” function type, usually supplied in the form of a lambda expression. That gets passed along to the `Lazy` property delegate. When `getValue()` is called on the `Lazy`, if it has not executed the initializer yet, it does so and uses that as the initial value to return from `getValue()`. It holds onto this value in its own property, and it uses that property for all other `getValue()` and `setValue()` calls.

Creating the Lazy itself is cheap. Only if you access the property and trigger the `getValue()` call on the Lazy will we wind up executing your lambda expression and incurring its expenses (CPU time, memory usage, etc.).

Other Stock Property Delegates

There are a few classes in the Kotlin standard library, besides Lazy, that are set up to serve as property delegates.

Map and MutableMap

Map has the `getValue()` function needed to serve as a `val` property delegate, so we can delegate a property to a Map:

```
val stuff = mapOf("something" to 1337, "somethingElse" to "like, whatever")
val something: Int by stuff
val somethingElse: String by stuff

println(something)
println(somethingElse)
```

(from ["Map as a Property Delegate" in the Klassbook](#))

Here, we use `by stuff` to delegate our properties to a Map named `stuff`. When we go to read those properties, the Map will look up the corresponding values, based on the property names, and return them.

Delegates

The Delegates class offers additional property delegates.

For example, `Delegates.observable()` allows you to provide a callback that will be invoked when the property value changes:

```
import kotlin.properties.Delegates

fun main() {
    var observed: String by Delegates.observable("initial value") { property, oldValue, newValue ->
        println("${property.name}' changed from '$oldValue' to '$newValue'")
    }

    println(observed)

    observed = "new value"
```

PROPERTY DELEGATES

```
println(observed)
}
```

(from "[Delegates.observable\(\)](#)" in the *Klassbook*)

Your callback receives the old and new property value, along with an object representing the property itself. That is a `KProperty` object, which we use here to get the property name.

There is also `vetoable()`:

```
import kotlin.properties.Delegates

fun main() {
    var noOddValuesPlease: Int by Delegates.vetoable(2) { property, oldValue, newValue ->
        newValue % 2 == 0
    }

    println(noOddValuesPlease)

    noOddValuesPlease = 4
    println(noOddValuesPlease)

    noOddValuesPlease = 3
    println(noOddValuesPlease)
}
```

(from "[Delegates.vetoable\(\)](#)" in the *Klassbook*)

This time, the lambda expression needs to return a `Boolean`. If it returns `true`, then the property value will be set to the new value. If it returns `false`, though, then the property value will be left alone. Therefore, the lambda expression can “veto” a change that it does not like. In this case, we veto any change that results in an odd number, giving us:

```
2
4
4
```

Delegating Properties to Properties

Kotlin 1.4 added a new feature: allowing you to delegate a property to another property.

Off the cuff, that may sound silly.

However, it is particularly useful for dealing with API changes, especially in situations where you cannot easily change the consumers of that API, such as a

PROPERTY DELEGATES

library that is reused by lots of projects.

Suppose, for example, that you decide that your mis-named a public property and want to switch to a different name. Perhaps you have:

```
class SomethingCool {
    var oldPropertyName: String = "default"
}
```

...and now you want:

```
class SomethingCool {
    var newPropertyName: String = "default"
}
```

The problem is that all existing users of your API are using the old name.

You can use property delegation to “rewire” the old API to really use your new property:

```
class SomethingCool {
    var newPropertyName: String = "default"

    var oldPropertyName by this::newPropertyName
}
```

Now, anything still using `oldPropertyName` still works.

You can combine this with the `@Deprecated` annotation to steer developers towards the new name:

```
class SomethingCool {
    var newPropertyName: String = "default"

    @Deprecated("Please use 'newPropertyName' instead", ReplaceWith("newPropertyName"))
    var oldPropertyName by this::newPropertyName
}
```

Here, we scope the reference using `this::` to refer to a property on the receiver itself. We could route it to some other object, if needed:

```
class SomethingCool(val otherCoolThing: SomethingElseCool) {
    var oldPropertyName by otherCoolThing::whatever
}
```


PROPERTY DELEGATES

Now, references to `oldPropertyName` on the `SomethingCool` wind up actually being handled by the whatever property on a `SomethingElseCool` object that was supplied to the `SomethingCool` constructor.

Class Delegation

Inheritance is a powerful construct in object-oriented programming... sometimes a bit *too* powerful. You hear the expression “favor composition over inheritance” a lot, because inheritance is fairly rigid:

- A class can only extend one other class
- A class gets *everything* that it inherits (excluding private elements)

The combination of those two limitations means that crafting a class hierarchy can be a challenge, so that only the *right* things are available in the *right* classes and not elsewhere.

Kotlin offers a spin on inheritance and composition that can help alleviate some of this, in the form of class delegation. Just as we saw [in the previous chapter](#) that you can use the `by` keyword to delegate a property, you can use `by` to delegate the implementation of an interface.

Playing Favorites

Suppose that we have some collection of items that we want to show the user, perhaps in some sort of a list. Items can be marked as “favorites”, akin to browser bookmarks, so we need to track which items are presently the favorites. However, for various reasons, the way that we update the favorites is different than how we work with the items themselves — for example, there might be different Web service endpoints for dealing with favorites. So we want to keep the favorite business logic a bit separate from the items themselves.

We could say that we have a `FavoriteStore` that tells us whether an item is marked as a favorite and to toggle that state for an item:

CLASS DELEGATION

```
interface FavoriteStore<T> {  
    fun isFavorite(thingy: T): Boolean  
  
    fun toggleFavorite(thingy: T, isFavorite: Boolean)  
}
```

Our actual Item class does not track its favorite status — we instead would get that from some instance of FavoriteStore:

```
data class Item(val id: String, val name: String)
```

Let's further suppose that this is being done in an Android app, one that has already adopted the Jetpack ViewModel system. So we have some sort of ViewModel that will handle displaying a collection of Item objects. However, that display not only includes the data from the Item itself, but also whether or not the Item is a favorite, so we can have an icon designating that status.

Somehow, consumers of our theoretical ItemViewModel will need to know whether or not a particular Item is a favorite or not, and we want that actual determination to be handled by some instance of FavoriteStore. That might be a memory-backed implementation for tests and a real one — saving to a database or Web service — for the real app.

We could go with something like this:

```
interface FavoriteStore<T> {  
    fun isFavorite(thingy: T): Boolean  
  
    fun toggleFavorite(thingy: T, isFavorite: Boolean)  
}  
  
class InMemoryFavoriteStore<T> : FavoriteStore<T> {  
    private val favorites: MutableMap<T, Boolean> = mutableMapOf()  
  
    override fun isFavorite(thingy: T) = favorites[thingy] ?: false  
  
    override fun toggleFavorite(thingy: T, isFavorite: Boolean) {  
        favorites[thingy] = isFavorite  
    }  
}  
  
data class Item(val id: String, val name: String)  
  
class ItemViewModel(val favorites: FavoriteStore<Item>)
```

CLASS DELEGATION

```
fun main() {
    val vm = ItemViewModel(InMemoryFavoriteStore())
    val item = Item("this is my id", "this is my name")

    println("item isFavorite: ${vm.favorites.isFavorite(item)}")
    vm.favorites.toggleFavorite(item, true)
    println("item isFavorite: ${vm.favorites.isFavorite(item)}")
}
```

We expose a favorites property from the ItemViewModel that is our FavoriteStore. Consumers of that ItemViewModel then work with the favorites object from there.

Manual Delegation

This works. However, arguably, it breaks encapsulation. The consumer of ItemViewModel perhaps should not have direct access to the FavoriteStore. Instead, ItemViewModel should have an API for working with favorites that hides the fact that there even is a FavoriteStore.

So, we could set up manual delegation, where ItemViewModel forwards the FavoriteStore API on to favorites:

```
class ItemViewModel(private val favorites: FavoriteStore<Item>) : FavoriteStore<Item>
{
    override fun isFavorite(thingy: Item) = favorites.isFavorite(thingy)

    override fun toggleFavorite(thingy: Item, isFavorite: Boolean) =
        favorites.toggleFavorite(thingy, isFavorite)
}

fun main() {
    val vm = ItemViewModel(InMemoryFavoriteStore())
    val item = Item("this is my id", "this is my name")

    println("item isFavorite: ${vm.isFavorite(item)}")
    vm.toggleFavorite(item, true)
    println("item isFavorite: ${vm.isFavorite(item)}")
}
```

This also works, and for a small interface like FavoriteStore, it is not *too* tedious to implement. The bigger the API surface, though, the more work is involved in setting up this manual delegation.

The Class Delegate Alternative

By using a class delegate, we get the best of both worlds: the `FavoriteStore` remains internal to the `ItemViewModel` implementation, yet we get a zero-cost implementation of the `FavoriteStore` API:

```
interface FavoriteStore<T> {
    fun isFavorite(thingy: T): Boolean

    fun toggleFavorite(thingy: T, isFavorite: Boolean)
}

class InMemoryFavoriteStore<T> : FavoriteStore<T> {
    private val favorites: MutableMap<T, Boolean> = mutableMapOf()

    override fun isFavorite(thingy: T) = favorites[thingy] ?: false

    override fun toggleFavorite(thingy: T, isFavorite: Boolean) {
        favorites[thingy] = isFavorite
    }
}

data class Item(val id: String, val name: String)

class ItemViewModel(favorites: FavoriteStore<Item>) : FavoriteStore<Item> by favorites

fun main() {
    val vm = ItemViewModel(InMemoryFavoriteStore())
    val item = Item("this is my id", "this is my name")

    println("item isFavorite: ${vm.isFavorite(item)}")
    vm.toggleFavorite(item, true)
    println("item isFavorite: ${vm.isFavorite(item)}")
}
```

(from "[Class Delegates](#)" in the *Klassbook*)

When we try using the `FavoriteStore` API on `ItemViewModel`, Kotlin forwards those calls along to `favorites`, without us having to do that delegation manually. Yet, `favorites` is just a constructor parameter — it is not even a property.

Now, in this case, we could also have `ItemViewModel` inherit from `InMemoryFavoriteStore()` and get the same result. However, in a real-world app, we will want to supply different `FavoriteStore` implementations, perhaps through dependency inversion frameworks. Forcing `ItemViewModel` to inherit from `InMemoryFavoriteStore` would eliminate the flexibility to swap in a different implementation for different circumstances.

Constants

A `val` property is immutable, but it is not a constant in Kotlin.

If that sounds confusing... that is not surprising.

Declaring a Constant

A constant is declared by adding the `const` keyword before `val`:

```
const val REQUEST_PERMISSIONS = 1337
```

And... that's it.

Why Bother, When We Have `val`?

`val` is immutable, but the value might not be determined until runtime:

```
import kotlin.random.Random  
val randomNumber = Random.nextInt(1, 100)
```

Here, we do not know what the value of `randomNumber` is until this property is initialized. We cannot change the value of `randomNumber`, but we do not know the value at compile time.

By contrast, `const` creates a compile-time constant.

Some bits of code need compile-time constants. The biggest one is annotations. An annotation property needs to be a compile-time constant, since the annotation itself

might be applied at the time the code is compiled.

Also, the Kotlin compiler may be able to perform additional optimizations for compile-time constants, since the value is known to the compiler, compared to simple immutable `val` values.

So, for things that can be a constant, using `const` has some value, though usually it is not essential.

Constant Type Limitations

However, you are going to find that not everything qualifies to be a constant:

- It has to be a primitive type (e.g., Boolean, Int, Long, Float) or a String
- It has to be either a top-level declaration or be declared in an object (including a companion object)
- You cannot [override the getter](#)
- You cannot use [property delegates](#)

Abstract Properties

Java developers who move to Kotlin will be used to [abstract classes](#) and abstract functions.

It is also possible to have abstract properties, odd as that may seem.

But, But, But... Why?

An abstract function declares a function signature that subclasses must implement (or themselves be abstract).

Similarly, an abstract property declares one or two function signatures that subclasses must implement:

- A `val` declares a getter
- A `var` declares a getter and a setter

The fact that this happens to be a property matters more to the caller, as they treat this as a perfectly normal property. From the standpoint of the subclasses, overriding the property is really there to provide implementations for the getter (and, where relevant, the setter).

Abstract `val`

Implementing an abstract `val` property is not significantly different than implementing a regular `val` property, other than needing the `override` keyword as you would with implementing an abstract function:

ABSTRACT PROPERTIES

```
abstract class Base {
    abstract val something: String
}

class SomethingSource : Base() {
    override val something: String = "like, whatever"
}

fun main() {
    println(SomethingSource().something)
}
```

(from "[Abstract val Properties](#)" in the *Klassbook*)

Here, Base defines an abstract property, and SomethingSource extends Base. As a result, SomethingSource needs to provide a concrete implementation of the abstract property. In this case, since it is a val, we just use a string literal.

This is equivalent to implementing [a custom getter](#):

```
abstract class Base {
    abstract val something: String
}

class SomethingSource : Base() {
    override val something: String
        get() = "like, whatever"
}

fun main() {
    println(SomethingSource().something)
}
```

However, the get() syntax provides more flexibility. For example, suppose that we wanted to use a random number instead of a string literal. If we assign a random number to something, that is the one-and-only something value:

```
import kotlin.random.Random

abstract class Base {
    abstract val something: Int
}

class SomethingSource : Base() {
    override val something = Random.nextInt(1,100)
}
```

ABSTRACT PROPERTIES

```
fun main() {
    val source = SomethingSource()

    println(source.something)
    println(source.something)
    println(source.something)
}
```

If, on the other hand, we implement `get()`, since `get()` is called on every access of the property, we get a fresh random number every time that we refer to `something`:

```
import kotlin.random.Random

abstract class Base {
    abstract val something: Int
}

class SomethingSource : Base() {
    override val something
        get() = Random.nextInt(1,100)
}

fun main() {
    val source = SomethingSource()

    println(source.something)
    println(source.something)
    println(source.something)
}
```

(from ["Abstract val Custom Getter" in the Klassbook](#))

Abstract var

Similarly, you can have an abstract var, but then the subclass needs to supply a getter and a setter:

```
abstract class Base {
    abstract var something: Int
}

class SomethingSource : Base() {
    override var something: Int = 0
        get() = field
        set(value) { field = value}
}
```

ABSTRACT PROPERTIES

```
}  
  
fun main() {  
    val source = SomethingSource()  
  
    println(source.something)  
    source.something = 1337  
    println(source.something)  
}
```

(from "[Abstract var](#)" in the *Klassbook*)

Covariance in Generics

Generics are powerful. Type inheritance is powerful. Sometimes, powerful things do not get along very well... and that can be the case with generics and inheritance.

In this chapter and [the next](#), we will explore why sometimes Kotlin complains about your generics, and how two simple keywords — in and out — can help solve the problem.

You Can't Put That in That!

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun main() {
    val animals: List<Animal> = listOf<Frog>(Frog(), Frog())

    println(animals)
}
```

Here, we have two sub-types of `Animal`: `Frog` and `Axolotl`. We create a `List<Animal>` variable named `animals` and try initializing it with a `List<Frog>`. Frogs are animals, and `Frog` is an `Animal`, so we would expect this to work... and it does.

However, simply switching from `List` to `MutableList` causes problems:

```
open class Animal

class Frog : Animal()
```

```
class Axolotl : Animal()

fun main() {
    val animals: MutableList<Animal> = mutableListOfOf<Frog>(Frog(), Frog())

    println(animals)
}
```

If you try this, you will get Type mismatch: inferred type is Animal but Frog was expected as a compile error.

If you look at [the Kotlin documentation for MutableList](#), you will see that it is declared as:

```
interface MutableList<E> : List<E>, MutableCollection<E>
```

MutableList extends the List and MutableCollection interfaces, using a generic type E.

[The Kotlin documentation for List](#), though, is a little bit strange:

```
interface List<out E> : Collection<E>
```

WTF is out?

out Is a Direction

out indicates that our dominant use of the generic type is to return values of that type. And, as such, we believe that it is safe for us to support sub-types of that type.

Remember that in Kotlin (and many strongly-typed object-oriented programming languages) a variable or property type need only be compatible with its value's actual type. Objects have type — a variable or property's type says “we are using this object *as if it its type were X*”, regardless of the actual type of that object.

So, suppose that MutableList allowed subtypes and this statement succeeded:

```
val animals: MutableList<Animal> = mutableListOfOf<Frog>(Frog(), Frog())
```

animals says “I hold a list of Animal objects”, and it allows you to assign any Animal to any index. So, from a compilation standpoint, this would work:

```
animals[0] = Axolotl()
```

After all, `Axolotl` is an `Animal`.

However, while `animals` says “I hold a list of `Animal` objects”, the actual *object* that `animals` points to says “I hold a list of `Frog` objects”. We cannot put an `Axolotl` in a `List<Frog>`, and so `animals[0] = Axolotl()` would cause one of two problems:

1. It would fail immediately, on the grounds that you cannot put an `Axolotl` in a `List<Frog>`
2. It would fail when we later try retrieving that element, because Kotlin would try treating an `Axolotl` as if it were a `Frog`, and that would result in a `ClassCastException` or similar problem

So, `MutableList` is *invariant* in its generic type. A `MutableList<Frog>` needs to be based on a list of `Frog` objects, and if we have a `MutableList<Animal>` variable or property, the actual underlying list needs to support any type of `Animal`, not just one sub-type.

But `List` does not support mutation. If you review the functions on `List`, either:

- They return the generic type, or
- They accept the generic type but only for comparison purposes (e.g., `contains()` to determine if the list contains a certain element)

In other words, for the operations that `List` performs, it does not matter if the underlying list is of a sub-type or not. All the `List` operations still work.

That is why `List` can use `out` to say that it is *covariant* in the generic type. A `List<Animal>` variable or property can point to a `List<Frog>` without issue, because at no point will we be forced to try to put an `Axolotl` into that `List`.

Declaration-Site and Use-Site Variance

Using `<out T>` in a type declaration, as `List` does, is called “declaration-site variance”. The alternative is “use-site variance” — using `<out T>` for a function parameter.

For example, let’s try to clone some frogs:

COVARIANCE IN GENERICS

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun <T> clone(input: MutableList<T>, output: MutableList<T>) {
    input.forEachIndexed { index, item -> output[index] = item}
}

fun main() {
    val input = mutableListOf(Frog(), Frog())
    val result = mutableListOf<Animal>(Axolotl(), Axolotl())

    clone(input, result)

    println(result)
}
```

`clone()` takes in two lists, and it assigns the values in the output list to be the same as the input list, on a per-index basis. This function should be performing some data validation, ensuring that the output list is the same length or longer than the input list, but we are skipping that to keep the example code shorter.

However, this code does not compile:

```
Type mismatch: inferred type is Animal but Frog was expected
```

The complaint is about the input parameters to `clone()`. We are passing into `clone()`:

- A `List<Frog>`, as that is what `mutableListOf(Frog(), Frog())` returns
- A `List<Animal>`, as that is what we specified `result` to be

However, `clone()` is expecting both parameters to be of type `T`, but we are passing in two different (but related) types: `Frog` and `Animal`.

Logically, what we want `clone()` to do is fine. A `Frog` is an `Animal`, so we can take a bunch of `Frog` objects and assign them to `Animal` slots in a list.

Adding out to the use site indicates that we can accept the input `MutableList` to be of type `T` or any sub-type:

COVARIANCE IN GENERICS

```
open class Animal

class Frog : Animal()

class Axolotl : Animal()

fun <T> clone(input: MutableList<out T>, output: MutableList<T>) {
    input.forEachIndexed { index, item -> output[index] = item}
}

fun main() {
    val input = mutableListOf(Frog(), Frog())
    val result = mutableListOf<Animal>(Axolotl(), Axolotl())

    clone(input, result)

    println(result)
}
```

That satisfies the compiler and makes logical sense: we can put an object of any `Animal` sub-type into an `Animal` slot in a list.

Contravariance in Generics

Continuing our examination of type inheritance and their impacts on generics... sometimes, we like to sort animals.

One option is to use `Comparable` and have `Animal` know how to compare itself to other `Animal` objects:

```
open class Animal : Comparable<Animal> {
    override fun compareTo(other: Animal) = toString().compareTo(other.toString())
}

class Frog : Animal() {
    override fun toString() = "Frog!"
}

class Axolotl : Animal() {
    override fun toString() = "Axolotl?"
}

fun main() {
    println(listOf(Frog(), Axolotl()).sorted())
}
```

`compareTo()` works as it does in Java: it needs to return:

- A negative number if the receiver sorts before the other value
- A positive number if the other value sorts before the receiver
- Zero if the two values are equal

This works well, sorting our `Axolotl` before our `Frog`:

```
[Axolotl?, Frog!]
```

CONTRAVARIANCE IN GENERICS

And, since our list contains a Frog and an Axolotl, Kotlin is going to find the common supertype — Animal — and treat our list as a List<Animal>.

We could also use Comparator, which has a compare() function that works like compareTo(). However, Comparator is a standalone interface, knowing how to compare two objects of some generic type:

```
open class Animal

class Frog : Animal() {
    override fun toString() = "Frog!"
}

class Axolotl : Animal() {
    override fun toString() = "Axolotl?"
}

class FrogComparator : Comparator<Frog> {
    override fun compare(one: Frog, two: Frog) =
        one.toString().compareTo(two.toString())
}

fun main() {
    println(listOf(Frog(), Frog()).sortedWith(FrogComparator()))
}
```

This works, because we are sorting a List<Frog> and FrogComparator knows how to compare frogs. But, suppose we pass in the same frogs... as a List<Animal>:

```
open class Animal

class Frog : Animal() {
    override fun toString() = "Frog!"
}

class Axolotl : Animal() {
    override fun toString() = "Axolotl?"
}

class FrogComparator : Comparator<Frog> {
    override fun compare(one: Frog, two: Frog) =
        one.toString().compareTo(two.toString())
}
```

CONTRAVARIANCE IN GENERICS

```
fun main() {  
    println(listOf<Animal>(Frog(), Frog()).sortedWith(FrogComparator()))  
}
```

Now, we have a compile error, because even though the underlying objects are of type `Frog`, we are treating them as `Animal`, and `FrogComparator` does not know how to compare `Animal` objects.

So, why can we sort a `List<Animal>` using `Comparable` but not `Comparator`? They are declared slightly differently, just as `List` and `MutableList` were in [the preceding chapter](#).

[Comparator works simply on a generic type T:](#)

```
fun interface Comparator<T>
```

...while [Comparable uses a new keyword, in](#):

```
interface Comparable<in T>
```

`in` indicates that our primary use of the data type is to accept it as input, in ways where we can work with sub-types for that input. In other words, `Comparable<Animal>` can compare a `Frog` or an `Axolotl`, as those are sub-types of `Animal`.

Using the plain generic type, as `Comparator` does, means that `Comparator` is *invariant* with respect to the type. Conversely, the use of `in` by `Comparable` means that `Comparable` is *contravariant* with respect to the type.

Declaration-Site and Use-Site Variance

As with `out`, `in` can be used in a declaration (as `Comparable` does) or at a use site (e.g., a function parameter).

For example, suppose that we want to replace a frog with an axolotl in a list:

```
open class Animal  
  
class Frog : Animal() {  
    override fun toString() = "Frog!"  
}
```

CONTRAVARIANCE IN GENERICS

```
class Axolotl : Animal() {
    override fun toString() = "Axolotl?"
}

fun replaceWithAnimal(list: MutableList<Animal>, value: Animal) {
    for (i in list.indices) {
        list[i] = value
    }
}

fun main() {
    val animals: MutableList<Any> = mutableListOf(Frog(), Frog())
    val replacement = Axolotl()

    replaceWithAnimal(animals, replacement)

    println(animals)
}
```

animals is a `MutableList<Any>`. So, even though it points to an object that contains only frogs, we are treating it as though it is a list that contains any possible type. This is a problem when we call `replaceWithAnimal()`, as it is expecting a `MutableList<Animal>`, not a `MutableList<Any>`.

The use of `in` changes that:

```
open class Animal

class Frog : Animal() {
    override fun toString() = "Frog!"
}

class Axolotl : Animal() {
    override fun toString() = "Axolotl?"
}

fun replaceWithAnimal(list: MutableList<in Animal>, value: Animal) {
    for (i in list.indices) {
        list[i] = value
    }
}

fun main() {
    val animals: MutableList<Any> = mutableListOf(Frog(), Frog())
    val replacement = Axolotl()

    replaceWithAnimal(animals, replacement)
}
```

CONTRAVARIANCE IN GENERICS

```
println(animals)
}
```

At a use site, it says “we accept this type or any supertype”. So, a `MutableList<Any>` satisfies `MutableList<in Animal>`. And, logically, this makes sense: we are attempting to replace an element of the list with an `Animal`, and that works whether the list is a list of `Animal` or a list of `Any`.

Note, though, that we are also relying on Kotlin’s type inference. `mutableListOf(Frog(), Frog())`, by default, would return a `MutableList<Frog>`. But we are not stating that explicitly. Kotlin sees that we are using it to initialize a variable of type `MutableList<Any>`, so it decides that we really wanted the list to actually *be* a `MutableList<Any>`. If we force it to be a `MutableList<Frog>`:

```
val animals: MutableList<Any> = mutableListOf<Frog>(Frog(), Frog())
```

...then we have a [covariance](#) problem. `MutableList` is invariant in its type, so we cannot assign a `MutableList<Frog>` to a `MutableList<Any>`.

And, if we turn around and say that `animals` is supposed to be a `MutableList<Frog>`:

```
val animals: MutableList<Frog> = mutableListOf(Frog(), Frog())
```

...then we have another covariance problem. `replaceWithAnimal()` will accept a `MutableList` of `Animal` or a supertype, but not sub-type. Hence, it will not accept a `MutableList<Frog>`, which makes sense, as we cannot assign an `Animal` to a slot in a list that only accepts `Frog`.

Anonymous Functions

What happens when the compiler is not smart enough to understand your lambda expression?

The Problem: Return Types

A lambda expression allows us to specify the types for any parameters, by adding the types in the parameter list:

```
val squarifier = { x: Int -> x * x }  
println(squarifier.invoke(3))
```

(from ["Lambda Expressions and Parameters" in the Klassbook](#))

Here, we specifically declare that `x` is of type `Int`.

However, we do not specify what the return type is. We can't — there simply is no syntax for it. Instead, the Kotlin compiler will infer the return type, based on what it knows about what we are returning.

Sometimes, though, Kotlin will not be able to infer what we want. For example, if we dropped the type from the above example:

```
fun main() {  
    val squarifier = { x -> x * x }  
    println(squarifier.invoke(3))  
}
```

...we get a compile error: `Cannot infer a type for this parameter. Please`

specify it explicitly.

However, there may be cases where the compiler is happy with the parameter types but cannot infer the return type, or it infers the wrong return type.

The Solution: Anonymous Function Syntax

A lambda expression is a “function literal”. The other syntax for function literals is the anonymous function:

```
val squarifier = fun(x: Int): Int { return x * x }  
  
println(squarifier.invoke(3))
```

(from "[Anonymous Functions](#)" in the *Klassbook*)

Here, `squarifier` has the same algorithm as before, just written in the form of an anonymous function. Here, we *have* to state our return type explicitly, just as we do for a regular function.

Syntactically, an anonymous function is a function declaration without the function name. We assign the anonymous function to a property, pass it as a parameter, etc., just as we would do for a lambda expression. And we execute the function the same way as we would do for a lambda expression, such as calling `invoke()` on it.

Constraints and Effects

Lambda expressions and anonymous functions can be used interchangeably for the most part. There are two notable differences, though.

Where You Can Supply Anonymous Functions

If a function takes a function type as the last parameter, we can pass a lambda expression outside of the parameter list parentheses:

```
data class Event(val id: Int)  
  
fun main() {  
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))  
  
    val leetEvent = events.first { it.id == 1337 }
```

ANONYMOUS FUNCTIONS

```
println(leetEvent)
}
```

(from "[first\(\)](#)" in the *Klassbook*)

Here, `first()` takes a function type, and we provide a lambda expression outside of the `first()` call parameter list. Since we have no other parameters on `first()`, we can even drop the parentheses from the `first()` call.

We lose those benefits with an anonymous function. We have to provide those inside the parentheses, the way that we would for any other type of parameter:

```
data class Event(val id: Int)

fun main() {
    val events = listOf(Event(1), Event(5), Event(1337), Event(24601), Event(42), Event(-6))

    val leetEvent = events.first(fun(it: Event): Boolean { return it.id == 1337 })

    println(leetEvent)
}
```

(from "[Anonymous Functions as Parameters](#)" in the *Klassbook*)

Where return Returns

A return can be [labeled](#) or without a label. A return without a label always returns from whatever function we are in.

An anonymous function is a function, but a lambda expression is not. Hence, the behavior of `return` is different between the two constructs. A `return` inside of an anonymous function returns from the anonymous function itself. A `return` inside of a lambda expression returns from whatever function encloses it.

Local Functions

We can have functions in classes and similar constructs, such as interfaces.

We can have top-level functions, outside of any class.

And, as it turns out, we can have local functions, where one function resides inside of another function.

I Heard You Like Functions...

A chunk of the Kotlin standard library is written as pure functions: top-level functions that are not part of any class. For these functions, the only input (ideally) comes as parameters.

Even more is written in the form of extension functions on existing classes. Extension functions *look* like what their name suggests: they extend an existing class. However, from a practical standpoint, extension functions really amount to “syntactic sugar” on top-level functions. An extension function has no ability to do anything that an equivalent top-level function cannot — the extension function simply gets to use `this` for one of the parameters. So, this:

```
fun String.withLength() = "$this ${this.length}"
```

...is equivalent to:

```
fun stringWithLength(s: String) = "$s ${s.length}"
```

Pure functions and extension functions have no state. All input is provided via parameters, where the extension function just has convenient syntax for supplying

one special parameter (`this`).

However, since these sorts of functions have no state, sub-dividing them into smaller functions can be a problem sometimes. You might have a really complex function where a piece of logic ought to be pulled out into a separate function, but both the original and the separate function share too many values.

You may also have cases where you want to pull logic out into a separate function, but that separate function has no purpose except when called in the context of its original function.

...So I Put a Function in Your Function

To help address those sorts of concerns, Kotlin supports local functions:

```
@Composable
private fun DrawerContent(
    navController: NavController,
    scaffoldState: ScaffoldState
) {
    fun navTo(route: String) {
        navController.navigate(route) {
            popUpTo = navController.graph.startDestination
            launchSingleTop = true
        }

        scaffoldState.drawerState.close()
    }

    Column {
        DRAWER_ITEMS.forEach {
            DrawerRow(it) { navTo(it.route) }
        }
    }
}
```

This is a bit of code using an alpha version of Jetpack Compose, a next-generation UI toolkit for Android. Without getting into all of the details, we have a `DrawerContent()` function that populates a navigation drawer that slides out from one edge of the screen. We want to have a list of rows in the drawer representing places you can navigate to in the app, where `DrawerRow()` renders a row. Part of the logic of `DrawerRow()` is to respond to click events on a row by calling the supplied lambda expression. There, we call `navTo()` to navigate to the desired portion of the

LOCAL FUNCTIONS

`app.navTo()` is implemented as a local function — its implementation appears inside of `DrawerContent()`.

`navTo()` could be a peer function of `DrawerContent()`. However, as it turns out, the only place that `navTo()` is needed is here in `DrawerContent()`. `DrawerContent()` is a top-level function; having `navTo()` *also* be a top-level function clutters up the top-level namespace. We could just have the body of `navTo()` go directly in the lambda expression provided to `DrawerRow()`... but we might have other navigation UI options besides rows, such as some buttons at the top of the drawer for popular destinations. Using a local `navTo()` function allows that logic to be reused as needed within `DrawerContent()`, but *only* within `DrawerContent()`, where it is needed.

Funception

There is no practical limit in how deep your local functions can nest. A function can have a local function, which has its own local function, and so on:

```
fun up() {
  fun you() {
    fun give() {
      fun gonna() {
        fun never() {

        }
      }
    }
  }
}
```

However, scoping rules apply. `up()` cannot call `never()`, as `never()` is defined local to `gonna()` — only `gonna()` can call `never()`.

Local Types

Not only can we [nest functions inside of functions](#), but we can also create local classes, interfaces, and other types as well!

Scenario: Perils of Pair Programming

Sometimes, we need some sort of data structure to represent an interim result.

Android developers using RxJava in Kotlin run into this scenario a lot, particularly with operators like `zip()`. `zip()` says “do this list of operations in parallel, then combine their results and give that to me”. `zip()` not only takes the operations (e.g., a pair of `Single` objects), but also some lambda expression or other function type that combines the results of each `Single` into a single object result:

```
Single.zip(userAuthCall, serverStatusCall) { userAuthResult, serverStatusResult ->
    TODO("ummmm... combine these... somehow...")
}
```

A cheap and easy solution is to use a `Pair`, which wraps an arbitrary pair of objects. The top-level `to()` [inline function](#) can create a `Pair`:

```
Single.zip(userAuthCall, serverStatusCall) { userAuthResult, serverStatusResult ->
    userAuthResult to serverStatusResult
}
```

Something later that subscribes to get the results will get the `Pair`:

```
Single.zip(userAuthCall, serverStatusCall) { userAuthResult, serverStatusResult ->
    userAuthResult to serverStatusResult
}
```


LOCAL TYPES

```
.subscribe { pairOfResults ->
    // TODO something cool
}
```

However, Pair just has first and second properties to get the wrapped values. first and second are fine property names, but they are very general and do not have any real meaning in our code.

One option is to use a [destructuring declaration](#) to break the Pair back apart into two variables:

```
Single.zip(userAuthCall, serverStatusCall) { userAuthResult, serverStatusResult ->
    userAuthResult to serverStatusResult
}
.subscribe { (userAuthResult, serverStatusResult) ->
    // TODO something cool
}
```

Another option would be a custom data class instead of Pair, so you can use property names that have greater meaning:

```
data class OpResults(val userAuth: UserAuth, val serverStatus: ServerStatus)

Single.zip(userAuthCall, serverStatusCall) { userAuthResult, serverStatusResult ->
    OpResults(userAuth = userAuthResult, serverStatus = serverStatusResult)
}
.subscribe { results ->
    // TODO something cool with results.userAuth and results.serverStatus
}
```

The question then becomes: where does that data class reside? If it has no value outside the boundaries of one function, does it make sense for this to be a top-level class or a nested class?

With Kotlin, you have another answer: declare the data class inside the function that needs it:

```
fun loadStuff() {
    data class OpResults(val userAuth: UserAuth, val serverStatus: ServerStatus)

    Single.zip(userAuthCall, serverStatusCall) { userAuthResult, serverStatusResult ->
        OpResults(userAuth = userAuthResult, serverStatus = serverStatusResult)
    }
    .subscribe { results ->
```

LOCAL TYPES

```
// TODO something cool with results.userAuth and results.serverStatus  
  }  
}
```

This is a bit more verbose than a `Pair`. However, ideally, if you use the `Pair`, you would have comments in the code explaining what `first` and `second` are. If you are going to bother explaining it via comments... perhaps you could explain it via code instead.

Inline Functions

Sometimes, you might find a function declared using the `inline` keyword:

```
inline fun max(x: Int, y: Int): Int = if (x > y) x else y

fun main() {
    val bigger = max(3, 7)

    println("The larger of 3 and 7 is $bigger")
}
```

(from "[Inline Functions](#)" in the *Klassbook*)

What is `inline`? What “line” are we “in”?

Macro History

Many programming languages — C and C++ among them — support preprocessor macros:

```
#define max(X, Y) ((X) > (Y) ? (X) : (Y))
```

Given the macro definition, you can use it like a regular function call:

```
foo = max(bar, goo);
```

As the name suggests, the “preprocessor” takes the raw source code and processes it before passing it to the actual compiler. In this case, the preprocessor expands the macro in place, replacing the above line with:

```
foo = ((bar) > (goo) ? (bar) : (goo));
```

As a result, we have syntax that looks like a function call, but we avoid some overhead involved with a function call versus just doing the work directly.

If we use the macro several times, though, there is a cost in terms of app size. We have the “expanded” macro code in several places, rather than it being just implemented in a single function body.

Inline Functions: Like Macros

An inline function in Kotlin works like a preprocessor macro. The compiler takes all occurrences of calls to that inline function and replaces them with the actual function body, much like how the C preprocessor replaces all occurrences of the macro with the macro definition.

What We Write

All we do is add `inline` to the function declaration:

```
inline fun max(x: Int, y: Int): Int = if (x > y) x else y

fun main() {
    val bigger = max(3, 7)

    println("The larger of 3 and 7 is $bigger")
}
```

(from ["Inline Functions" in the Klassbook](#))

What the Compiler Compiles

The compiler does not generate an actual function, despite our declaration. Rather, each place that we call the function, Kotlin will compile the code from the function body. It will be as if we wrote:

```
fun main() {
    val bigger = if (3 > 7) 3 else 7

    println("The larger of 3 and 7 is $bigger")
}
```

We gain some performance by avoiding an actual function call. But, if we call `max()` in many places, we wind up with many copies of the `max()` code in the compiled

app, and our app will be bigger.

Inline Properties

If you implement a property with [custom accessors](#) that do not reference the backing field, those accessors can be marked as `inline`:

```
val stuff = mutableMapOf<String, String?>("something" to "This is the 'something' value")
var something: String?
    inline get() = stuff["something"]
    inline set(value) { stuff["something"] = value }

fun main() {
    something = "This is different"
    println(something)
}
```

And, if you want both accessors to be `inline` — as shown above — you can mark the whole property as `inline`:

```
val stuff = mutableMapOf<String, String?>("something" to "This is the 'something' value")
inline var something: String?
    get() = stuff["something"]
    set(value) { stuff["something"] = value }

fun main() {
    something = "This is different"
    println(something)
}
```

(from "[Inline Properties](#)" in the *Klassbook*)

Inline Classes

With Kotlin 1.3, `inline` isn't just for functions anymore!

What?

In [the preceding chapter](#), we saw `inline` applied to functions. This says “instead of making the actual function call, put the code for the function ‘inline’ where the call is being made”.

Kotlin 1.3 added support for `inline` applied to classes:

```
inline class InvoiceLineItemKey(val key: String)
```

An `inline class` is only allowed to have one property, and it must appear in the constructor. They are allowed to have functions and properties [with custom accessors](#) (but no backing fields).

When the code is compiled — assuming that there are no compile errors — all occurrences of this `inline class` are replaced by references to the one-and-only property of that class, and all references to function calls are replaced with inline representations of those functions.

Why?

One key use case for this will be with object-relational mapping engines (ORMs) and similar systems, for helping to avoid “stringly-typed” properties.

Suppose, for example, that we have a database that has invoices, where invoices have a series of line items representing the things to appear on the invoice. In Kotlin, we

INLINE CLASSES

might have `Invoice` and `InvoiceLineItem` classes to map to the tables in our database. And those classes might have properties that line up with the columns in the table... using data types based on the column types:

```
data class Invoice(  
    val key: String,  
    val createdOn: Instant,  
    val customerKey: String,  
    // TODO other properties  
)  
  
data class InvoiceLineItem(  
    val key: String,  
    val quantity: Int,  
    val productKey: String,  
    // TODO other properties  
)
```

The problem here is that all of our keys are `String` properties:

- the key on `Invoice`
- the key on `InvoiceLineItem`
- the key to our customer, perhaps in a `Customer` class and table
- the key to our product, perhaps in a `Product` class and table

From Kotlin's standpoint, a `String` is a `String` is a `String`. We as developers know that we cannot use an `InvoiceLineItem` key to look up a product... but Kotlin does not know that, so it cannot enforce such a restriction. This sort of “stringly-typed” set of properties leads to bugs, where we accidentally use one class' key where we meant another class' key, and Kotlin lets it happen because they are all strings.

With `inline class`, we can wrap those strings in classes that are distinct:

```
data class Invoice(  
    val key: InvoiceKey,  
    val createdOn: Instant,  
    val customerKey: CustomerKey,  
    // TODO other properties  
)  
  
data class InvoiceLineItem(  
    val key: InvoiceLineItemKey,  
    val quantity: Int,
```

INLINE CLASSES

```
val productKey: ProductKey,  
    // TODO other properties  
)
```

Each of those `...Key` classes is an `inline class` akin to the one shown earlier in this chapter:

```
inline class InvoiceLineItemKey(val key: String)
```

At compile time, Kotlin treats all four of those `...Key` classes as being distinct and not equivalent. We cannot accidentally use a `ProductKey` where we need an `InvoiceKey`, for example. This eliminates a class of bugs through strong type checking in the compilation process.

However, because they are `inline class`, at *runtime*, they are just strings. We do not incur overhead of creating instances of all of these `...Key` classes. So, we gain the power of type checking without paying a price when our app runs.

Alpha!

Note, though, that `inline class` is an alpha feature at the present time. You need to [enable it](#) as part of your build options before using it. And there may be bugs, as with any alpha-grade software.

Reified Type Parameters

“Reify” is a word in English, meaning:

to consider or represent something abstract as a material or concrete thing;
to give definite content and form to a concept or idea

(from [the Merriam-Webster dictionary](#))

From a programming standpoint, [the early January 2021 edition of Wikipedia](#) has:

By means of reification, something that was previously implicit, unexpressed, and possibly inexpressible is explicitly formulated and made available to conceptual (logical or computational) manipulation. Informally, reification is often referred to as “making something a first-class citizen” within the scope of a particular system.

Despite the flowery descriptions, the reified keyword in Kotlin has a very specific meaning: keep a generic type around for use at runtime.

Type Erasure (Other Than Via the Backspace Key)

We saw the use of generic types [earlier in the book](#). One key aspect of generic types is that they are purely something used at compile time. The Kotlin compiler uses generic types to help ensure the validity of our code, yelling at us if we violate some contract. However, the *compiled code* has no knowledge of those generic types.

For example, collections like `List` have a `filterIsInstance()` operator. This returns another collection whose types match a particular type. We could try to define our own similar function, `filterByType()`:

REIFIED TYPE PARAMETERS

```
fun <T> List<*>.filterByType() = this.filter { it is T }

fun main() {
    println(listOf(1, "this", 3, "is", "a", 1337, "mess").filterByType<String>())
}
```

Here, `filterByType()` is declared as an extension function of `List<*>`, where the wildcard means “we are not concerned about the type of objects in this `List`”. The function declares a generic type `T`, and the implementation of the function uses the regular `filter()` operator with a lambda that returns true only for those objects that are of type `T`.

In theory, this should work.

In practice — such as if you try this in the [Klassbook scratch pad](#) — it fails with a compile error:

```
Cannot check for instance of erased type: T
```

The `is` check is a runtime check — we do not know at compile time what is in the `List` courtesy of the wildcard. But that implies that at runtime we know what type `T` is, and by default, we do not, because types get “erased” by the compilation process. Usually, our attempts to use a generic type at runtime get detected by the compiler, and we get an error message like the one shown above.

Reified = Retained Type for Inline Functions

For smaller functions, the workaround is to make [the function itself be inline](#) while also adding the `reified` keyword to the generic type:

```
inline fun <reified T> Iterable<*>.filterByType() = this.filter { it is T }

fun main() {
    val raw = listOf(1, "this", 3, "is", "a", 1337, "mess")
    val filtered = raw.filterByType<String>()

    println(filtered)
}
```

(from ["Reified Types" in the Klassbook](#))

This tells the compiler to do two things:

REIFIED TYPE PARAMETERS

1. As with all `inline` functions, the actual implementation should be placed into each call site, so while it *looks* like we are calling a function, in reality we are directly executing a copy of the function body
2. In that copy of the function body, convert `T` into the actual type used by that particular call

In this case, we know that we want `T` to be `String`, because that is how we are calling `filterByType()`, using `filterByType[String]()`. `reified` says that we take the `T` in the inline function and replace it by the actual type. So, it is as if we had written the code as:

```
fun main() {
    val raw = listOf(1, "this", 3, "is", "a", 1337, "mess")
    val filtered = raw.filter { it is String }

    println(filtered)
}
```

The `filterByType()` implementation is “inlined” where we make the call, and the concrete type for `T` is also inlined.

This is one of those keywords that you will tend to only use when the compiler yells at you, or *maybe* you encounter some runtime error. Any time you get an error related to type erasure, that suggests that you should be using `inline` and `reified`.

`inline` is really designed for small functions, where the overhead of having a bunch of copies of the function body is not too bad. If you get type erasure errors on a larger function, you will want to try to isolate the bit that has the actual type erasure problem, refactor that into a separate function, then `inline` that function and use `reified` to clear up the type erasure issue.

noinline and crossinline

Most likely, `noinline` and `crossinline` are keywords that you will add to your code when the Kotlin compiler tells you to add them to your code by way of an error message.

For many developers, simply doing what the compiler tells them to do is sufficient. However, in case you really want to know *why* the compiler is complaining... read on!

noinline: Keep the Lambda as an Object

When we mark a function as `inline`, the code associated with that function is used as a replacement for any calls to that function, so we avoid the overhead of the function call. Most of the time, we focus on “the code associated with that function” as being the function body. However, it *also* means the code associated with any lambda expressions passed as function type parameters to the function.

Take this code, for example:

```
inline fun beUseful(whenDone: () -> Unit) {
    // TODO something useful

    whenDone()
}

fun main() {
    beUseful {
        println("hello, world!")
    }
}
```

Because `beUseful()` is marked as `inline`, not only is the *body* of `beUseful()` inlined

at the call site, but so is the lambda expression that we pass as `whenDone()`.

Sometimes, this is fine — all we want to do with the function type is `invoke()` it, as we are doing here. However, in other cases, we will try to treat the function type as an object, storing it somewhere or passing it to some other function:

```
fun somethingElse(thingToDo: () -> Unit) {
    thingToDo()
}

inline fun beUseful(whenDone: () -> Unit) {
    // TODO something useful

    somethingElse(whenDone)
}

fun main() {
    beUseful {
        println("hello, world!")
    }
}
```

Here, we get:

```
Illegal usage of inline-parameter 'whenDone' in 'public inline fun beUseful(whenDone:
() -> Unit): Unit
defined in root package in file klassbook.kt'. Add 'noinline' modifier to the
parameter declaration
```

So, we can follow the directions of the compiler and add `noinline` to that parameter:

```
fun somethingElse(thingToDo: () -> Unit) {
    thingToDo()
}

inline fun beUseful(noinline whenDone: () -> Unit) {
    // TODO something useful

    somethingElse(whenDone)
}

fun main() {
    beUseful {
        println("hello, world!")
    }
}
```

This tells Kotlin to not treat that parameter as inline, so it is able to be treated as an object and be passed around.

crossinline: Allowing return

You might have funky code that tries doing a return from inside of a lambda expression:

```
fun beUseful(whenDone: () -> Unit) {
    // TODO something useful

    whenDone()
}

fun main() {
    beUseful {
        println("hello, world!")
        return
    }

    println("...and we're done!")
}
```

This fails to compile:

```
'return' is not allowed here
```

You cannot return from the middle of some lambda expression.

However, if you mark the function as inline:

```
inline fun beUseful(whenDone: () -> Unit) {
    // TODO something useful

    whenDone()
}

fun main() {
    beUseful {
        println("hello, world!")
        return
    }

    println("...and we're done!")
}
```

NOINLINE AND CROSSINLINE

The code compiles... but ...and we're done! does not get printed. Effectively, what the inline keyword does is turn that code into:

```
fun main() {
    println("hello, world!")
    return

    println("...and we're done!")
}
```

As a result, we return from main() before we reach the second println() call.

If you want to block this sort of return from being used, add crossinline to the function type parameter declaration:

```
inline fun beUseful(crossinline whenDone: () -> Unit) {
    // TODO something useful

    whenDone()
}

fun main() {
    beUseful {
        println("hello, world!")
        return
    }

    println("...and we're done!")
}
```

Now, we are back to the original 'return' is not allowed here compiler error.

Receivers in Function Types

[Earlier in the book](#), we saw the `apply()` scope function:

```
class SomethingOrAnother {
    var someProperty = 123
}

val foo = SomethingOrAnother().apply {
    someProperty = 456
}
```

In the lambda expression that we pass to `apply()`, the value of `this` is whatever we call `apply()` upon. `with()` works similarly, except that `this` is whatever we pass as the parameter to `with()`:

```
class SomethingOrAnother {
    var someProperty = 123
}

fun main() {
    val foo = SomethingOrAnother()

    with(foo) {
        someProperty = 456
    }
}
```

You might think that this is some sort of compiler magic. In a sense, perhaps it is. However, both `apply()` and `with()` are functions implemented in Kotlin itself, and you can use this approach to tailor this in some context of your own.

What with() Looks Like

with() is a very simple [inline function](#):

```
public inline fun <T, R> with(receiver: T, block: T.() -> R): R {  
    return receiver.block()  
}
```

It takes two parameters, receiver and block. receiver is just an ordinary parameter, albeit one declared using a generic type T. block is where the fun lies.

with() uses two generic types: T and R. T is the type for the receiver parameter, and R is what with() returns.

block is declared as T.() -> R. () -> R is a declaration of a [function type](#). T.() -> R is a declaration of an *extension* function type, where we need to associate any call of that function type with something of type T.

with() invokes block via receiver.block(). The combination of T.() -> R syntax and receiver.block() says that this, in the context of the invoked block, is receiver.

If we go back to the with() example from earlier in this chapter:

```
class SomethingOrAnother {  
    var someProperty = 123  
}  
  
fun main() {  
    val foo = SomethingOrAnother()  
  
    with(foo) {  
        someProperty = 456  
    }  
}
```

receiver in this case is our SomethingOrAnother instance (foo). block is our lambda expression ({ someProperty = 456 }). Given how with() is implemented, this in that lambda expression is the SomethingOrAnother instance, which is why we can manipulate someProperty without specifying that object directly.

What apply() Looks Like

apply() is similar, though it is written as an extension function on T itself:

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

Since apply() is an extension function of T, it can just call block() directly. This is already the correct receiver: the same object that apply() was called upon.

The other major difference is that with() returns the results of receiver.block(), while apply() returns this (the receiver).

Use Case: DSL

Kotlin is well-suited for crafting domain-specific languages (DSLs). You can use the “extension function on a receiver” trick to offer small DSLs to simplify the use of some reusable bit of code.

For example, Android developers are starting to use [Jetpack Compose](#), and its Compose UI subset, for defining user interfaces. Compose UI uses little DSLs in several places to help reduce the amount of code needed for simple scenarios.

This code snippet sets up a vertically-scrolling list:

```
LazyColumn {
    items(content.forecasts) {
        WeatherRow(it)
    }
}
```

The items() call inside of LazyColumn() provides the list of model objects that represent the data for the list — in this case, as list of weather forecasts for a particular location. The lambda expression passed to items() gets called as needed to render each item in the list, based on the number of items, if the user scrolls, etc.

The trailing lambda parameter on LazyColumn() is defined as:

```
content: LazyListScope.() -> Unit
```

RECEIVERS IN FUNCTION TYPES

`LazyColumn()`, as part of its implementation, creates a `LazyListScope` object and uses that for invoking the content parameter (`scope.apply(content)`).

`LazyListScope` defines the API for the DSL: the functions that will be available to that content lambda:

```
interface LazyListScope {
    fun item(key: Any? = null, content: @Composable LazyItemScope.() -> Unit)

    fun items(
        count: Int,
        key: ((index: Int) -> Any)? = null,
        itemContent: @Composable LazyItemScope.(index: Int) -> Unit
    )
}
```

`LazyColumn()`, in turn, will use the `items()` supplied by that lambda as part of rendering the list.

This is a fairly common pattern in Compose UI: a function that you call takes a trailing lambda, with the receiver set to be something useful for that trailing lambda to define what it is supposed to do. In the case of functions like `LazyColumn()`, `LazyRow()`, `ConstraintLayout()`, and `NavHost()`, the receiver is set to some object that implements an interface that represents the DSL, so we can concisely tell Compose UI what it needs to do.

Renamed Imports

What happens when you love two classes so much that you cannot bear to be apart from either of them... but they share a name?

The Scenario: Observable

We have a few classes and interfaces named `Observable` in Android app development:

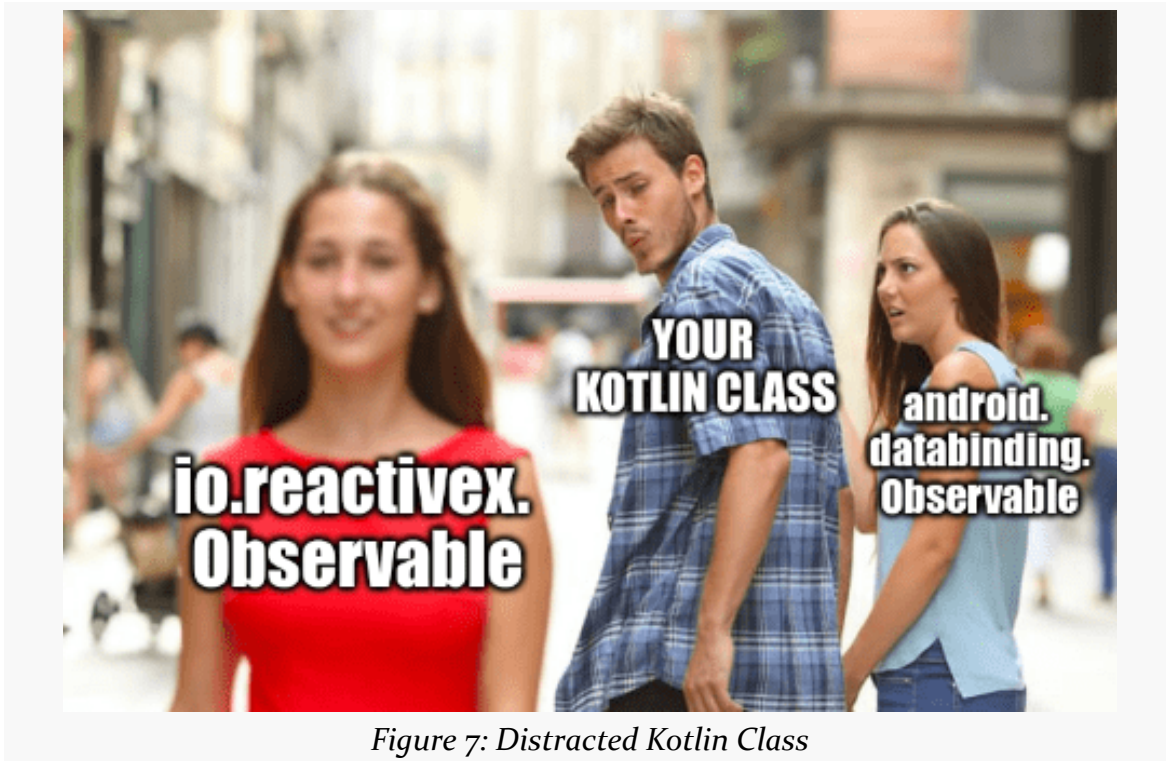
- `java.util.Observable`
- `android.database.Observable`
- `android.databinding.Observable`
- `io.reactivex.Observable` (for those using RxJava 2)
- Perhaps ones from other libraries

(and this does not even count Kotlin's own `observable()`, which at least is all lowercase)

In most cases, we only need one of these in any given Kotlin class.

The Problem: Import Name Collisions

But what if we need two?



In Java, the solution is clunky: only one can have an `import` statement, and the other has to be fully-qualified. So, you have to pick a favorite `Observable` that gets the short name, with references to the other one cluttering up the code.

The Solution: `import ... as ...`

Kotlin has a simple solution for this: import aliases. Akin to how we use `as` for [type aliases](#), we can use `as` to assign an alias to an import:

```
import android.databinding.Observable
import io.reactivex.Observable as RxObservable
```

Now, we can use `RxObservable` instead of `io.reactivex.Observable`.

This can be a bit confusing, as casual code reviewers might not notice the import

RENAMED IMPORTS

alias and might not recognize the alias name. However, particularly when you use an IDE for working with Kotlin, finding the source of the alias usually is easy enough, and the alias usually is less cumbersome than is the fully-qualified class name.

Operator Overloading

In other words, what do you get when you divide a `String` by an `Int`? And, can you change that behavior?

What the #@\$&%!?

Many programming languages involve math. That math usually involves symbols serving as operators:

```
val thisIsNotQuitePi = 22 / 7
```

Here, `/` is a symbol representing the division operator.

The Concept of Operator Overloading

Somewhere, the programming language needs to know that `/` means division, and it needs to actually implement the division functionality.

Worse, the division logic may vary somewhat by data type. Dividing two integers might be handled somewhat differently than would dividing two floating-point numbers.

Some languages simply “bake in” their support for operators and their corresponding operations. Java, for example, considers operators and what they do to be part of the core language.

Other languages — such as Kotlin — support [operator overloading](#). Simply put:

- The language defines a mapping between an operator and a function name

- (e.g., / mapping to a `divide()` function)
- The language's standard library offers implementations of the function for expected data types (e.g., `divide()` on `Number`)
- The language allows developers to implement the same-named function on other classes and gain operator support for those classes (e.g., `divide(Int)` on `String` to support splitting a string into a specified number of pieces)

Some languages — such as Swift — take the next step and allow developers to define brand-new operators (e.g., `***`), which then get treated the same as the standard operators. Kotlin does not go to that length, though.

Example: Dividing a String

By default, in Kotlin, this fails with a compile error:

```
fun main() {
    val pieces = "This is a reasonably long string" / 3
}
```

Kotlin does not know how to divide a `String` by an `Int`. However, we can teach Kotlin to do this, by way of implementing a `div()` [extension function](#) on `String`:

```
operator fun String.div(count: Int) = this.chunked((this.length / count) + 1)

fun main() {
    println("This is a reasonably long string" / 3)
}
```

(from "[Operator Overloading](#)" in the *Klassbook*)

Just as we need to override a function that is declared in a supertype, we need to add the operator keyword to a function whose name is used by Kotlin as the implementation of an operator. Here, `div()` is the function that Kotlin maps `/` to. So, our `div()` extension function allows us to divide a `String` by an `Int`.

So. Many. Operators.

[There are many operators in Kotlin](#), including some things that you might not expect to be considered operators.

in

`in` is an operator. It maps to the `contains()` function. So, any type that supports `contains()` — `String`, `List`, etc. — can be used with `in`:

```
fun main() {
    println("foo" in "foobar")
}
```

While `in` happens to be an operator, we can create other “operators” that syntactically look like `in`. For example, we used to `in` `mapOf()`:

```
val things = mapOf("key" to "value", "other-key" to "other-value")

println(things::class)
println(things)
println(things.size)
```

(from ["Maps" in the Klassbook](#))

`to` is not an operator. Instead, it is an [infix function](#), and we can create our own infix functions.

Indexed Access

Square-bracket syntax gets mapped to operators. `[]` maps to `get()`, while `[]=` maps to `set()`.

In the case of `get()`, the comma-delimited list of arguments in the brackets gets passed to the `get()` function. As a result, we can use `[]` for both a single index — like we do with `List` or `Map` — and for multiple indexes.

In the case of `set()`, the comma-delimited list of arguments in the brackets gets passed to the `set()` function, followed by the value to the right of the operator. So, `something['foo'] = 3` is equivalent to `something.set('foo', 3)`.

Invocation

When we see syntax like `foo()`, we assume that `foo` is the name of a function. Most of the time, that is the case. However, it could also be an object whose type has an `invoke()` function:

OPERATOR OVERLOADING

```
operator fun String.invoke(count: Int) = this.chunked((this.length / count) + 1)

fun main() {
    println("This is a reasonably long string"(3))
}
```

(from ["invoke\(\) Operator" in the Klassbook](#))

Here we have the same function that we had for `div()`, except now it is named `invoke()`. We call it by “calling a function” on the object, passing an `Int` parameter.

Infix Functions

As [the chapter on operator overloading](#) mentioned, you cannot implement your own operators... at least not in the form of punctuation. And, we saw that some operators, like `in`, are not based on punctuation.

The nice thing is that Kotlin lets you create your own “operators”, to support syntax like `"this is very impolite".disemvowel "*"`.

Postfix and Infix

Fans of classic HP calculators will remember “reverse Polish notation” (RPN) as the data entry format. To calculate 2 plus 2, you would press `2` `Enter` `2` `+` on the keyboard. The `Enter` key pushes a number onto the stack, and the `+` operation takes the current number, pops the last number off of the stack, and adds them. This is called “postfix” notation.

Calculators from other manufacturers often use an `=` key. There, to calculate 2 plus 2, you would press `2` `+` `2` `=`. This “infix” notation is also how most programming languages work (Forth being one exception).

The `infix` Keyword

If you add the `infix` keyword to a function declaration, you can use that function using infix notation, skipping the parentheses and looking more like the `in` operator:

```
private val VOWELS = "[aeiouAEIOU]".toRegex()
infix fun String.disemvowel(replacement: String) = VOWELS.replace(this, replacement)
```


INFIX FUNCTIONS

```
fun main() {  
    println("this is very impolite" disemvowel "")  
}
```

(from "[Infix Functions](#)" in the *Klassbook*)

To “[disemvowel](#)” some text is simply to remove the vowels, replacing them with placeholders or just eliminating them outright. Here, we have a `disemvowel()` extension function on `String` that removes the English-language vowels (ignoring “y” and “Y” because English is weird). You are welcome to call `disemvowel()` using regular function notation:

```
println("this is very impolite".disemvowel(""))
```

However, the `infix` keyword means that we can use this function much like we would use an operator like in:

- No dot connecting the function name to the object on which we are calling it (the “receiver”)
- No parentheses around the parameter

Regardless of whether you use `disemvowel()` as a function or as a pseudo-operator, you get a disemvoweled string:

```
th*s *s v*ry *mp*1*t*
```

Limitations

The `infix` function needs to take exactly one parameter. This means:

- No zero-parameter functions
- No functions with 2+ parameters
- No varargs
- No default values

Also, it cannot be a top-level function — the function needs to have a receiver of some form. In the preceding example, we have it as an extension function, though a regular class function is also fine.

Destructuring Declarations

A Kotlin expression might well return more than one value. This does not mean “more than one value, but all stored in a single object”, though obviously Kotlin can do that too. Instead, this means literally getting more than one result from the expression:

```
val (something, orAnother) = gimmeStuff()
```

Here, `gimmeStuff()` will populate both `something` and `orAnother`... even though `gimmeStuff()` has only one return value.

This trick is called a “destructuring declaration”.

The Components of a Class

For this to work, the object returned by the expression (e.g., the object returned by `gimmeStuff()`) needs to implement functions named `component1()`, `component2()`, and so on. The parenthetical list of properties will use those functions in numerical order. So, the above code snippet is equivalent to:

```
val temp = gimmeStuff()
val something = temp.component1()
val orAnother = temp.component2()
```

(of course, this second snippet also sets up a `temp` variable that the first snippet avoids)

Many built-in classes, like `Pair`, implement these functions, one for every distinct piece of data. So, in the case of `Pair`, it implements `component1()` and `component2()`, but not others, as a `Pair` only holds two items.

DESTRUCTURING DECLARATIONS

Also, data classes implement these functions, with the numbered functions corresponding with the constructor parameter positions:

```
data class Person(  
    val name: String,  
    val quest: String,  
    val airSpeedVelocityUnladenSwallow: Float  
)  
  
fun main() {  
    val (who, what, waitWut) = Person("Arthur", "To seek the Holy Grail", 0.01f)  
    println(who)  
    println(what)  
    println(waitWut)  
}
```

(from ["Destructuring Declarations" in the Klassbook](#))

Here, our Person class winds up with three component functions:

Component Function	Returned Property
component1()	name
component2()	quest
component3()	airSpeedVelocityUnladenSwallow

(note: the unit of measurement for airSpeedVelocityUnladenSwallow is in [furlongs per microfortnight](#))

OK, Why Would We Use This?

Well, in general, you probably should not use it. It will be rather fragile as classes change. For example, if somebody changes Person to be:

```
data class Person(  
    val name: String,  
    val quest: String,  
    val favoriteColor: Color,  
    val airSpeedVelocityUnladenSwallow: Float  
)
```

DESTRUCTURING DECLARATIONS

Now `waitWut` winds up being set to the `favoriteColor` value, not the `airSpeedVelocityUnladenSwallow` value. In this case, the fact that `waitWut` becomes a `Color` instead of a `Float` means that you might wind up with compiler errors and will detect the fact that `waitWut` changed. But, if instead `Person` were changed to:

```
data class Person(  
    val name: String,  
    val capitalOfAssyria: String,  
    val quest: String,  
    val airSpeedVelocityUnladenSwallow: Float  
)
```

Now the value of `wait` will change, but the type will not, so we might not catch the difference.

(if you do not get the odd references here, go watch [this movie](#), preferably several times)

So, Why Does This Exist?

If the author had to guess, destructuring declarations were added for use when iterating over a `Map`:

```
fun main() {  
    val montyPythonCast = mapOf(  
        "Graham Chapman" to "Arthur, King of the Britons",  
        "John Cleese" to "Sir Lancelot",  
        "Terry Gilliam" to "Patsy",  
        "Eric Idle" to "Sir Robin",  
        "Terry Jones" to "Sir Bedevere",  
        "Michael Palin" to "Sir Galahad"  
    )  
  
    for ((actor, role) in montyPythonCast) {  
        println("$actor -> $role")  
    }  
}
```

(from "[Destructuring Declarations of Maps](#)" in the *Klassbook*)

Normally, a for loop over a `Map` gives you a single `Map.Entry` property, and you have to refer to the key and value on that to get to the individual pieces of data that you are looking for. With destructuring declarations, you can have the for loop give you the key and value in individual properties that have useful names (rather than key

DESTRUCTURING DECLARATIONS

and value).

This also works with lambda expressions, such as using `forEach()` on a `Map`:

```
fun main() {
    val montyPythonCast = mapOf(
        "Graham Chapman" to "Arthur, King of the Britons",
        "John Cleese" to "Sir Lancelot",
        "Terry Gilliam" to "Patsy",
        "Eric Idle" to "Sir Robin",
        "Terry Jones" to "Sir Bedevere",
        "Michael Palin" to "Sir Galahad"
    )

    montyPythonCast.forEach { (actor, role) -> println("$actor -> $role") }
}
```

(from "[Destructuring Declarations and Lambdas](#)" in the *Klassbook*)

Labeled Returns

Sometimes, in Kotlin, you will see a return followed by @ and some name, such as `return@overthere`. This is a “labeled return” and allows you to have some control over where return actually returns to.

Where return Goes By Default

Frequently, there is little debate over the behavior of return:

```
fun theFunction() {  
    return  
}
```

Here, return returns from `theFunction()`.

Things get messier when we start to introduce nested constructs, such as lambda expressions:

```
fun somethingifier(items: List<String>) {  
    items.forEach {  
        if (it.length == 3) return else println(it)  
    }  
  
    println("Done!")  
}  
  
fun main() {  
    somethingifier(listOf("this", "is", "a", "fun", "bit", "of", "Kotlin"))  
}
```

(from ["Default return Behavior" in the Klassbook](#))

LABELED RETURNS

Here, we return if we encounter a three-letter word. You might think that this would just exit the lambda expression, so we would print “Done!”. However, by default, return returns from whatever *function* it is in. Lambda expressions are not functions. So, our return returns entirely from somethingifier(), bypassing the “Done!” println() statement. So, we get:

```
this
is
a
```

While a lambda expression is not a function, an [anonymous function](#) is a function. Rewriting the above snippet to use an anonymous function changes the behavior a bit:

```
fun somethingifier(items: List<String>) {
    items.forEach(fun(it: String) {
        if (it.length == 3) return else println(it)
    })

    println("Done!")
}

fun main() {
    somethingifier(listOf("this", "is", "a", "fun", "bit", "of", "Kotlin"))
}
```

(from ["return and Anonymous Functions" in the Klassbook](#))

Now, our “business logic” is contained in an anonymous function, not a lambda expression. return, by default, returns from whatever function it is in, so our return returns from the anonymous function. That just completes the current pass in the forEach loop and allows the loop to continue. As a result, we just skip the three-letter words, and we get “Done!” at the end:

```
this
is
a
of
Kotlin
Done!
```

Returning Just from a Lambda

If you want the second set of results, but you want the simpler syntax of a lambda expression, you can use a labeled return:

```
fun somethingifier(items: List<String>) {
    items.forEach {
        if (it.length == 3) return@forEach else println(it)
    }

    println("Done!")
}

fun main() {
    somethingifier(listOf("this", "is", "a", "fun", "bit", "of", "Kotlin"))
}
```

(from ["Labeled Returns" in the Klassbook](#))

The function that invokes a lambda expression sets an implicit label, of the same name as the function itself. Here, `forEach()` is invoking the lambda expression, so we have a `forEach` label that we can use. `return@forEach`, therefore, returns from just the lambda expression, not from the entire `somethingifier()` function.

Applying a Label

The problem with implicit labels comes when you have nested calls of the same function type (e.g., a `forEach()` whose lambda expression contains a `forEach()`).

You can explicitly label a lambda expression if you like, then use `return@` with your custom label.

A label is an identifier plus `@`, preceding the lambda expression itself:

```
fun somethingifier(items: List<String>) {
    items.forEach toSender@ {
        if (it.length == 3) return@toSender else println(it)
    }

    println("Done!")
}
```


LABELED RETURNS

```
fun main() {  
    somethingifier(listOf("this", "is", "a", "fun", "bit", "of", "Kotlin"))  
}
```

(from ["Custom Labels" in the Klassbook](#))

So, here, we have labeled the lambda expression to `toSender@`, so we can use `return@toSender` to return from just the lambda expression.

Note that `break` and `continue` also support labels, both implicit and custom ones, to help control where they go to.

Is Any of This a Good Idea?

You do not see labels and labeled returns used much in Kotlin code samples. Their use may be a sign of overly-complex logic within a single function. Consider decomposing the work into separate functions, if they let you avoid labeled returns. The resulting code is likely to be more readable, particular for relative newcomers to Kotlin.

Nothing

Earlier in the book, we saw the `TODO()` function:

```
fun heyThisIsNotDoneYet() {  
    TODO("implement this function")  
}
```

`TODO()` throws an exception with the supplied message. We use this as a more forceful alternative to simply using a `TODO` comment to identify incomplete work.

Contrast that with:

```
fun heyThisIsNotDoneYet(): Int {  
    TODO("implement this function")  
}
```

On the surface, this would seem like it could not possibly compile. We have set up `heyThisIsNotDoneYet()` to return an `Int`, yet we have no return statement or anything else to return a value. After all, if we skipped the `TODO()`, we definitely would get a compilation error:

```
fun heyThisIsNotDoneYet(): Int {  
    // TODO implement this function  
}
```

So, what is it about `TODO()` that allows it to somehow complete the body of a function that returns something, while not returning anything?

The answer is: nothing. Or, more accurately, `Nothing`.

Nothing: It's On the Bottom

Earlier in the book, we saw `Any`. `Any` in Kotlin is analogous to `Object` in Java: it is the root type from which all other types inherit. In truth, `Any` is even more widespread in Kotlin than `Object` is in Java, because *all* Kotlin types extend `Any`, whereas primitives (e.g., `int`) and arrays (e.g., `int<>`) do not extend `Object` in Java.

`Nothing` has no Java analogue and is on the far other end of the inheritance spectrum. Effectively, `Nothing` is a sub-type of *all other types*. It is a sub-type of `Any`. It is a sub-type of `String`. It is a sub-type of `Axolotl`.

`Nothing` has no instances. It is impossible to create instances of `Nothing`, as it has a private constructor.

Uses of Nothing

On the surface, `Nothing` may seem useless. And, in day-to-day development, it is unlikely that you will use `Nothing` directly... but it is very likely that you are using `Nothing` without realizing it.

As a Return Type

A function can return `Nothing` as a type. This might seem impossible, since there are no instances of `Nothing`. From a compiler standpoint, though, a function returning `Nothing` means the function *is required to never return*. The primary scenario of this is if the function is guaranteed to throw an exception.

That may sound strange... until you consider `TODO()`.

`TODO()` is declared to return `Nothing`:

```
public inline fun TODO(reason: String): Nothing =  
    throw NotImplementedError("An operation is not implemented: $reason")
```

`TODO()` is marked as being [an inline function](#), so the compiler will “bake” the exception right into wherever the `TODO()` appears. However, the `Nothing` return type indicates to the compiler that since `TODO()` can never return, there is no sense in worrying about anything else in the function after that point, including the missing return.

NOTHING

For example, suppose we created our own similar function, called NOTDONE():

```
public inline fun NOTDONE(reason: String) {
    throw NotImplementedError("$reason")
}

fun heyThisIsNotDoneYet(): Int {
    NOTDONE("wut")
}

fun main() {
    heyThisIsNotDoneYet()
}
```

This fails compilation with A 'return' expression required in a function with a block body ('{...}') for our heyThisIsNotDoneYet() function. Even though NOTDONE() is inline, and so heyThisIsNotDoneYet() will always throw an exception, the compiler is not quite smart enough to figure that out on its own. Adding Nothing as the return type to NOTDONE() clears that up:

```
public inline fun NOTDONE(reason: String): Nothing {
    throw NotImplementedError("$reason")
}

fun heyThisIsNotDoneYet(): Int {
    NOTDONE("wut")
}

fun main() {
    heyThisIsNotDoneYet()
}
```

(from "[Nothing As a Return Type](#)" in the *Klassbook*)

On the Right Side of Elvis

TODO() is not the only function that returns Nothing. error() does as well:

```
public inline fun error(message: Any): Nothing = throw
IllegalStateException(message.toString())
```

error() forms nice shorthand for throwing an exception based on a failed null check using the Elvis operator:

NOTHING

```
fun main() {
    val something: String? = "foo"
    val somethingNotNull = something ?: error("hey, that was null!")

    println(somethingNotNull)
}
```

The reason why this compiles is that `Nothing` is a sub-type of all types, including `String`. Hence, from a type-safety standpoint:

- The left side of the Elvis operator is a `String`, because we know that it is not `null`
- The right side of the Elvis operator is `Nothing`
- The compiler finds the common supertype of those, which is `String`, and considers the expression's overall type to be `String`

For Covariant Generics

The `out` keyword signifies [covariance](#) in a generic type: we can accept the type or any sub-type. `List`, for example, uses `out`:

```
interface List<out E> : Collection<E>
```

As a result, we can use a `List<Nothing>` in place of any other typed `List`: `List<String>`, `List<Axolotl>`, etc.

The same holds true for any covariant generic type.

For Generic Singletons

That previous section might seem esoteric. After all, if there are no instances of `Nothing`, we certainly cannot have a `List` of such instances. However, just because a `List<Nothing>` is impossible does not mean that a `List<Nothing>` has no uses.

For example, suppose we need an empty list of something. One way to do that is to use `listOf()` with no contents:

```
val things: List<String> = listOf()
```

An alternative is to use `emptyList()`:

```
val things: List<String> = emptyList()
```

NOTHING

Those look nearly identical, but they actually have significantly different implementations. `listOf()` will instantiate an empty `List`, allocating a bit of memory along the way. `emptyList()` does not... because `emptyList()` returns a singleton:

```
public fun <T> emptyList(): List<T> = EmptyList
```

...and `EmptyList` is a `List<Nothing>`:

```
internal object EmptyList : List<Nothing>, Serializable, RandomAccess {
    private const val serialVersionUID: Long = -7390468764508069838L

    override fun equals(other: Any?): Boolean = other is List<*> && other.isEmpty()
    override fun hashCode(): Int = 1
    override fun toString(): String = "[]"

    override val size: Int get() = 0
    override fun isEmpty(): Boolean = true
    override fun contains(element: Nothing): Boolean = false
    override fun containsAll(elements: Collection<Nothing>): Boolean =
elements.isEmpty()

    override fun get(index: Int): Nothing =
        throw IndexOutOfBoundsException("Empty list doesn't contain element at index
$index.")
    override fun indexOf(element: Nothing): Int = -1
    override fun lastIndexOf(element: Nothing): Int = -1

    override fun iterator(): Iterator<Nothing> = EmptyIterator
    override fun listIterator(): ListIterator<Nothing> = EmptyIterator
    override fun listIterator(index: Int): ListIterator<Nothing> {
        if (index != 0) throw IndexOutOfBoundsException("Index: $index")
        return EmptyIterator
    }

    override fun subList(fromIndex: Int, toIndex: Int): List<Nothing> {
        if (fromIndex == 0 && toIndex == 0) return this
        throw IndexOutOfBoundsException("fromIndex: $fromIndex, toIndex: $toIndex")
    }

    private fun readResolve(): Any = EmptyList
}
```

From a practical standpoint, both `listOf()` and `emptyList()` fill the role of an empty list, but because `emptyList()` returns a singleton, `emptyList()` does not allocate memory.

NOTHING

From a compilation standpoint, `emptyList()` can be applied to any type, because `Nothing` is a sub-type of any type.

Types of Keywords

Many programmers are used to the notion that you cannot use keywords as the names of variables, properties, functions, and the like. So, for example, this fails to compile:

```
fun main() {  
    val class = 1337  
  
    println(class)  
}
```

But some keywords, like `import`, work just fine:

```
fun main() {  
    val import = 1337  
  
    println(import)  
}
```

There are a few types of keywords in Kotlin, and each has its rules for where and when you can use them as identifiers.

Hard Keywords

Hard keywords like `class`, cannot be used as identifiers anywhere. Most Kotlin keywords are considered to be hard.

[The roster of hard keywords](#) will vary by Kotlin version, but it includes things that you might expect beyond `class`, such as:

- `else`

- false
- fun
- if
- interface
- null
- object
- package
- return
- this
- true
- try
- val
- var
- when

Soft Keywords

A [soft keyword](#) has a meaning in a particular context, and otherwise is available for use as an identifier.

For example, `try` appeared in the above list of hard keywords. `catch` and `finally` do not, and not because the author did not supply the complete list. `catch` and `finally` are considered to be soft keywords, as is `import`, which means that you can use them as identifiers in most places.

“Most places”, though, may not be everywhere.

So, for example, `constructor` is a soft keyword. You can have a variable named `constructor`, if you wanted:

```
fun main() {
    val constructor = 1337

    println(constructor)
}
```

However, at least on Kotlin/JS, you cannot have a function named `constructor()`:

```
class Foo {
    fun constructor() {
        println("I'm not a constructor!")
    }
}
```

TYPES OF KEYWORDS

```
}  
  
fun main() {  
    Foo().constructor()  
}
```

In this case, the problem appears to be tied to JavaScript interoperability, as the Kotlin/JS transpiler generates a function named `constructor()` that is the actual constructor, so having your own `constructor()` function causes a name collision.

Modifier Keywords

A [modifier keyword](#) also has a meaning in a specific context. In this case, the context is a declaration, and the keyword can be used elsewhere as an identifier.

For example, `public` is a modifier keyword, so you could have a variable named `public` if you wanted:

```
fun main() {  
    val public = 1337  
  
    println(public)  
}
```

You can even have a function named `public`:

```
class Foo {  
    fun public() {  
        println("I am public, and I am public()")  
    }  
}  
  
fun main() {  
    Foo().public()  
}
```

There may be select areas where you cannot use modifier keywords as identifiers, but on the whole, you can.

Other notable modifier keywords include:

- `abstract`
- `companion`

- `const`
- `data`
- `enum`
- `lateinit`
- `open`
- `override`
- `private`
- `protected`
- `sealed`
- `vararg`

In General, Avoid Keywords

Can you be cute and have a property named `public`? Can you be even more cute and have a private property named `public`?

Yes.

Is this a good idea?

Probably not.

Remember that the point behind identifiers is to tell all developers — both your current teammates and “future you” — what the variable, property, function, etc. means. Reusing keywords for this purpose can cause confusion, even if the syntax will be completely legitimate and safe.

Frequently, the keyword makes for a poor identifier anyway. `set` is a soft keyword, but you can still use it as a variable name:

```
fun main() {  
    val set = setOf(1, 3, 3, 7)  
  
    println(set)  
}
```

You might think “well, it is a `Set`, so why not call it `set`?” However, what it does not convey is what it is a `Set of`. Any `Set` could be named `set` — there is no context for which of your many `Set` objects this `set` happens to refer to.

Just because the compiler allows certain syntax does not mean that the choice of

TYPES OF KEYWORDS

syntax is good. Write code that is easy to read.

