



2ND EDITION

Kotlin Programming

THE BIG NERD RANCH GUIDE

Andrew Bailey, David Greenhalgh
& Josh Skeen

Kotlin Programming: The Big Nerd Ranch Guide

by **Andrew Bailey, David Greenhalgh and Josh Skeen**

Copyright © 2021 Big Nerd Ranch, LLC

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, contact

Big Nerd Ranch, LLC
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
<http://www.bignerdranch.com/>
book-comments@bignerdranch.com

The 10-gallon hat with propeller logo is a trademark of Big Nerd Ranch, LLC.

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
<http://www.informit.com>

The authors and publisher have taken care in writing and printing this book but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

ISBN-10 0136870481
ISBN-13 978-0136870487

Second edition, first printing, September 2021
Release E.2.1.1

Dedication

To Mom and Dad. Of all the words in this book, the ones that do justice to the support and mentorship you've given me over the years were the hardest to find. These two will have to suffice: Thank you.

— A.B.

To Rebecca, a driven, patient, beautiful woman, and the reason that this book came to be. To Mom and Dad, for valuing education above all else.

— D.G.

For Baker, the best little bug.

— J.S.

Acknowledgments

Although our names appear on the cover, a book is not made by its authors alone. Many people have contributed to the creation of this book. We owe our thanks to everyone who helped make this book what it is today:

- Bryan Sills, Michael Yotive, Nate Sottek, Jeremy Sherman, and Mark Duran, for graciously lending your eyes and offering feedback on updates for the second edition of this book.
- Eric Maxwell, for teaching an early version of the second edition and providing feedback about our coroutines, channels, and flows chapters.
- Loren Klingman and Jake Sower, for your feedback on the Kotlin/JS chapter.
- Drew Fitzpatrick, for reading through an early version of the second edition in its entirety and for your feedback on our Kotlin Multiplatform and Kotlin/Native content.
- Liv Vitale, Christian Keur, Zachary Waldowski, and David House, for constantly offering your expertise on other platforms, which helped greatly with our Kotlin/JS and Kotlin/Native discussions. Thank you for tolerating the deluge of oddly specific questions and for making us look smarter than we really are.
- Javontay McElroy, our talented Big Nerd Ranch designer who made the print book's IntelliJ IDEA cheat sheet. Thank you for enthusiastically diving head-first into the unfamiliar territory of printed material and for making this design from scratch.
- Eric Wilson, Madison Witzler, Franklin O'Neal, and CJ Best, who together are the masterminds of Big Nerd Ranch's training department. Our classes – and, by extension, this book – could not happen without the hard work you do.
- Our editor, Elizabeth Holaday. When we told you our plans for the second edition of this book, the first thing you said was, “So this is a pretty substantial update.” You would, of course, turn out to be absolutely correct in that assessment. Thank you for all your hard work along this journey to refine this book, amplify its strengths, and shore up its weaknesses.
- Simone Payment, our copyeditor and proofreader. Thank you for your help in putting the finishing touches on this book.
- Ellie Volckhausen, who designed our front cover. Surf's up!
- Chris Loper at IntelligentEnglish.com, who designed and produced the print and digital versions of this book. We also extensively leveraged his DocBook toolchain.
- Aaron Hillegass and Stacy Henry. As a practical matter, it would not be possible for this book to exist without Big Nerd Ranch, the company Aaron founded and Stacy now fearlessly leads.

Finally, thank you to all our students. Being your teacher offers us the opportunity to be a student in many ways, and for that we are immensely grateful. Teaching is part of the greatest thing that we do, and it has been a pleasure working with you. We hope that the quality of this book matches your enthusiasm and determination.

Table of Contents

[Introducing Kotlin](#)

[Why Kotlin?](#)

[Who Is This Book For?](#)

[How to Use This Book](#)

[For the More Curious](#)

[Challenges](#)

[Typographical conventions](#)

[Using an eBook](#)

[Looking Forward](#)

[I. Getting Started](#)

[1. Your First Kotlin Application](#)

[Installing IntelliJ IDEA](#)

[Your First Kotlin Project](#)

[Creating your first Kotlin file](#)

[Running your Kotlin file](#)

[The Kotlin REPL](#)

[For the More Curious: Why Use IntelliJ?](#)

[For the More Curious: Targeting the JVM](#)

[Challenge: REPL Arithmetic](#)

[2. Variables, Constants, and Types](#)

[Types](#)

[Declaring a Variable](#)

[Kotlin's Built-In Types](#)

[Read-Only Variables](#)

[Type Inference](#)

[Compile-Time Constants](#)

[Inspecting Kotlin Bytecode](#)

[For the More Curious: Java Primitive Types in Kotlin](#)

[Challenge: hasSteed](#)

[Challenge: The Unicorn's Horn](#)

[Challenge: Magic Mirror](#)

[II. Basic Syntax](#)

[3. Conditionals](#)

[if/else Statements](#)

[Adding more conditions](#)

[Nested if/else statements](#)

[More elegant conditionals](#)

[Ranges](#)

[when Expressions](#)

[when expressions with variable declarations](#)

[when expressions without arguments](#)

[Challenge: Trying Out Some Ranges](#)

[4. Functions](#)

[Extracting Code to Functions](#)

[Anatomy of a Function](#)

[Function header](#)

[Function body](#)

[Function scope](#)

[Calling a Function](#)

[Writing Your Own Functions](#)

[Default Arguments](#)

[Single-Expression Functions](#)

[Unit Functions](#)

[Named Function Arguments](#)

[For the More Curious: The Nothing Type](#)

[For the More Curious: File-Level Functions in Java](#)

[For the More Curious: Function Overloading](#)
[For the More Curious: Function Names in Backticks](#)

[5. Numbers](#)

[Numeric Types](#)
[Integers](#)
[Floating Point Numbers](#)
[Formatting a Double](#)
[Converting Between Numeric Types](#)
[For the More Curious: Unsigned Numbers](#)
[For the More Curious: Bit Manipulation](#)

[6. Strings](#)

[String Interpolation](#)
[Raw Strings](#)
[Reading Console Input](#)
[Converting Strings to Numbers](#)
[Regular Expressions](#)
[String Manipulation](#)
[Strings are immutable](#)
[String Comparison](#)
[For the More Curious: Unicode](#)

[7. Null Safety and Exceptions](#)

[Nullability](#)
[Kotlin's Explicit Null Type](#)
[Compile Time vs Runtime](#)
[Null Safety](#)
[Option one: checking for null values with an if statement](#)
[Option two: the safe call operator](#)
[Option three: the non-null assertion operator](#)
[Exceptions](#)
[Throwing an exception](#)
[Handling exceptions](#)
[Try/catch expressions](#)
[Preconditions](#)
[For the More Curious: Custom Exceptions](#)
[For the More Curious: Checked vs Unchecked Exceptions](#)

[III. Introduction to Functional Programming and Collections](#)

[8. Lambda Expressions and the Function Type](#)

[Introducing NyetHack](#)
[Anonymous Functions](#)
[Lambda Expressions](#)
[The function type](#)
[Implicit returns](#)
[Function arguments](#)
[The it identifier](#)
[Accepting multiple arguments](#)
[Type Inference Support](#)
[More Effective Lambdas](#)
[Defining a Function That Accepts a Function](#)
[Trailing lambda syntax](#)
[Function Inlining](#)
[Lambdas and the Kotlin Standard Library](#)
[For the More Curious: Function References](#)
[For the More Curious: Capturing Lambdas](#)
[Challenge: New Titles and Moods](#)

[9. Lists and Sets](#)

[Lists](#)
[Accessing a list's elements](#)
[Changing a list's contents](#)
[Iteration](#)
[Reading a File into a List](#)

[Destructuring](#)

[Sets](#)

[Creating a set](#)

[Adding elements to a set](#)

[while Loops](#)

[Collection Conversion](#)

[For the More Curious: Array Types](#)

[For the More Curious: Read-Only vs Immutable](#)

[For the More Curious: The break Expression](#)

[For the More Curious: Return Labels](#)

[Challenge: Formatted Tavern Menu](#)

[Challenge: Advanced Formatted Tavern Menu](#)

[10. Maps](#)

[Creating a Map](#)

[Accessing Map Values](#)

[Adding Entries to a Map](#)

[Modifying Map Values](#)

[Converting Between Lists and Maps](#)

[Iterating Through a Map](#)

[Challenge: Complex Orders](#)

[11. Functional Programming Basics](#)

[Transforming Data](#)

[map](#)

[associate](#)

[Destructuring with functional programming](#)

[flatMap](#)

[Filtering Data](#)

[filter](#)

[Combining Data](#)

[zip](#)

[Why Functional Programming?](#)

[Sequences](#)

[For the More Curious: Profiling](#)

[For the More Curious: Aggregating Data](#)

[reduce](#)

[fold](#)

[sumBy](#)

[For the More Curious: The vararg Keyword](#)

[For the More Curious: Arrow.kt](#)

[Challenge: Reversing the Values in a Map](#)

[Challenge: Finding the Most-Liked Menu Item](#)

[12. Scope Functions](#)

[apply](#)

[let](#)

[run](#)

[with](#)

[also](#)

[takeIf](#)

[Using Scope Functions](#)

[IV. Object-Oriented Programming](#)

[13. Classes](#)

[Defining a Class](#)

[Constructing Instances](#)

[Class Functions](#)

[Visibility and Encapsulation](#)

[Class Properties](#)

[Property getters and setters](#)

[Property visibility](#)

[Computed properties](#)

[Using Packages](#)

[For the More Curious: A Closer Look at var and val Properties](#)

[For the More Curious: Guarding Against Mutability](#)

[For the More Curious: Package Private](#)

[14. Initialization](#)

[Constructors](#)

[Primary constructors](#)

[Defining properties in a primary constructor](#)

[Secondary constructors](#)

[Default arguments](#)

[Named arguments](#)

[Initializer Blocks](#)

[Initialization Order](#)

[Delaying Initialization](#)

[Late initialization](#)

[Lazy initialization](#)

[For the More Curious: Initialization Gotchas](#)

[For the More Curious: Property Delegates](#)

[Challenge: The Riddle of Excalibur](#)

[15. Inheritance](#)

[Defining the Room Class](#)

[Creating a Subclass](#)

[Type Checking](#)

[The Kotlin Type Hierarchy](#)

[Type casting](#)

[Smart casting](#)

[Refactoring the Tavern](#)

[For the More Curious: Any](#)

[For the More Curious: The Safe Cast Operator](#)

[16. Objects, Data Classes, and Enums](#)

[The object Keyword](#)

[Object declarations](#)

[Object expressions](#)

[Companion objects](#)

[Nested Classes](#)

[Data Classes](#)

[toString](#)

[equals and hashCode](#)

[copy](#)

[Destructuring declarations](#)

[Enumerated Classes](#)

[Operator Overloading](#)

[Exploring the World of NyetHack](#)

[For the More Curious: Defining Structural Comparison](#)

[For the More Curious: Algebraic Data Types](#)

[For the More Curious: Value Classes](#)

[Challenge: More Commands](#)

[Challenge: Implementing a World Map](#)

[Challenge: Ring the Bell](#)

[17. Interfaces and Abstract Classes](#)

[Defining an Interface](#)

[Implementing an Interface](#)

[Default Implementations](#)

[Abstract Classes](#)

[Combat in NyetHack](#)

[Challenge: Additional Monsters](#)

[V. Advanced Kotlin](#)

[18. Generics](#)

[Defining Generic Types](#)

[Generic Functions](#)

[Generic Constraints](#)

[in and out](#)

[Adding Loot to NyetHack](#)

[For the More Curious: The reified Keyword](#)

[19. Extensions](#)

[Defining Extension Functions](#)

[Defining an extension on a superclass](#)

[Generic extension functions](#)

[Operator extension functions](#)

[Extension Properties](#)

[Extensions on Nullable Types](#)

[Extensions, Under the Hood](#)

[Extension Visibility](#)

[Extensions in the Kotlin Standard Library](#)

[For the More Curious: Function Literals with Receivers](#)

[Challenge: Frame Extension](#)

[20. Coroutines](#)

[Blocking Calls](#)

[Enabling Coroutines](#)

[Coroutine Builders](#)

[Coroutine Scopes](#)

[Structured Concurrency](#)

[Using an HTTP Client](#)

[async and await](#)

[For the More Curious: Race Conditions](#)

[For the More Curious: Server-Side Kotlin](#)

[Challenge: No Cancellations](#)

[21. Flows](#)

[Setting Up a Flow](#)

[MutableStateFlow](#)

[Flow Termination](#)

[Flow Transformations](#)

[Error Handling in Flows](#)

[For the More Curious: SharedFlow](#)

[22. Channels](#)

[Dividing Work with Channels](#)

[Sending to a Channel](#)

[Receiving from a Channel](#)

[Closing a Channel](#)

[Joining Jobs](#)

[For the More Curious: Other Channel Behaviors](#)

[Rendezvous channels](#)

[Buffered channels](#)

[Unlimited channels](#)

[Conflated channels](#)

[VI. Interoperation and Multiplatform Applications](#)

[23. Java Interoperability](#)

[Interoperating with a Java Class](#)

[Interoperability and Nullity](#)

[Type Mapping](#)

[Getters, Setters, and Interoperability](#)

[Beyond Classes](#)

[Exceptions and Interoperability](#)

[Function Types in Java](#)

[24. Introduction to Kotlin Multiplatform](#)

[What Is Kotlin Multiplatform?](#)

[Planning Your Multiplatform Project](#)

[Your First Multiplatform Project](#)

[Defining a Kotlin/JVM Target](#)

[Defining Shared Code](#)

[expect and actual](#)

[25. Kotlin/Native](#)

[Declaring a macOS Target](#)

[Writing Native Code with Kotlin](#)

[Launching a Kotlin/Native application](#)

[Kotlin/Native Outputs](#)

[For the More Curious: Kotlin Multiplatform Mobile](#)

[For the More Curious: Other Native Platforms](#)

[26. Kotlin/JS](#)

[Declaring Support for Kotlin/JS](#)

[Interacting with the DOM](#)

[The external Keyword](#)

[Executing Raw JavaScript](#)

[Dynamic Types](#)

[For the More Curious: Front-End Frameworks](#)

[Challenge: Currency Exchange Fees](#)

[27. Afterword](#)

[Where to Go from Here](#)

[Shameless Plugs](#)

[Thank You](#)

[Glossary](#)

[Index](#)

Introducing Kotlin

In 2011, JetBrains announced the development of the Kotlin programming language, an alternative to writing code in languages like Java or Scala to run on the Java Virtual Machine. Six years later, Google announced that Kotlin would be an officially supported development path for the Android operating system.

Kotlin's scope quickly grew from a language with a bright future into the language powering applications on the world's foremost mobile operating system. Today, large companies like Google, Uber, Netflix, Capital One, Amazon, and more have embraced Kotlin for its many advantages, including its concise syntax, modern features, and seamless interoperability with legacy Java code.

Why Kotlin?

To understand the appeal of Kotlin, you first need to understand the role of Java in the modern software development landscape. The two languages are closely tied, because Kotlin code is most often written for the Java Virtual Machine.

Java is a robust and time-tested language and has been one of the most commonly written languages in production codebases for years. However, since Java was released in 1995, much has been learned about what makes for a good programming language. Java is missing the many advancements that developers working with more modern languages enjoy.

Kotlin benefits from the learning gained as some design decisions made in Java (and other languages, like Scala) have aged poorly. It has evolved beyond what was possible with older languages and has corrected what was painful about them. You will learn more in the coming chapters about how Kotlin improves on Java and offers a more reliable development experience.

And Kotlin is not just a better language to write code to run on the Java Virtual Machine. It is a multiplatform language that aims to be general purpose: Kotlin can be used to write native macOS, iOS, and Windows applications; JavaScript applications; and, of course, Android applications. Recently, JetBrains has been investing in these cross-platform capabilities; Kotlin Multiplatform offers a unique way to share code across applications and has led to an increase in Kotlin's use beyond the Java Virtual Machine.

Who Is This Book For?

We have written this book for developers of all kinds: experienced Android developers who want modern features beyond what Java offers, server-side developers interested in learning about Kotlin's features, developers looking to share Kotlin code between their native or web apps, and newer developers looking to venture into a high-performance compiled language.

Android support might be why you are reading this book, but the book is not limited to Kotlin programming for Android. In fact, all the Kotlin code in this book is agnostic to the Android framework. That said, if you are interested in using Kotlin for Android application development, this book shows off some common patterns that make writing Android apps a breeze in Kotlin.

Although Kotlin has been influenced by a number of other languages, you do not need to know the ins and outs of any other language to learn Kotlin. From time to time, we will discuss the Java code equivalent for Kotlin code you have written. We will also point out similarities to other languages as they are relevant. If you have experience in these languages, this will help you understand the relationship between Kotlin and the platforms it supports. Even if these comparisons are less familiar to you, seeing how another language tackles the same problems can help you grasp the principles that have shaped Kotlin's development.

How to Use This Book

This book is not a reference guide. Our goal is to guide you through the most important parts of the Kotlin programming language. You will be working through example projects, building knowledge as you progress. To get the most out of this book, we recommend that you type out the examples in the book as you read along. Working through the projects will help build muscle memory and will give you something to carry on from one chapter to the next.

Also, each chapter builds on the topics presented in the last, so we recommend that you do not jump around. Even if you feel that you are familiar with a topic in other languages, we suggest that you read straight through – Kotlin handles many problems in unique ways. You will begin with introductory topics like variables and control flow, work your way through object-oriented and functional programming techniques, try out Kotlin’s first-party approach to running asynchronous code, and dip your toes in Kotlin’s Multiplatform capabilities. By the end of the book, you will have built your knowledge of Kotlin from that of a beginner to a more advanced developer.

Having said that, do take your time: Branch out, use the Kotlin reference at kotlinlang.org/docs/reference to follow up on anything that piqued your curiosity, and experiment.

For the More Curious

Most of the chapters in this book have a section or two titled “For the More Curious.” Many of these sections illuminate the underlying mechanisms of the Kotlin language. The examples in the chapters do not depend on the information in these sections, but they provide additional information that you may find interesting or helpful.

Challenges

Many chapters end with one or more challenges. These are additional problems to solve that are designed to further your understanding of Kotlin. We encourage you to give them a try to enhance your Kotlin mastery.

We recommend making a copy of your projects before working on a challenge; other chapters will often build on previous solutions and you will not want your changes to get in the way. You can also download the solutions for the exercises in this book at bignerdranch.com/kotlin-2e-solutions/.

Typographical conventions

As you build the projects in this book, we will guide you by introducing a topic and then showing how to apply your newfound knowledge. For clarity, we stick to the following typographical conventions.

Variables, values, and types are shown with fixed-width font. Class, function, and interface names are given bold font.

All code listings are shown in fixed-width font. If you are to type some code in a code listing, that code is denoted in bold. If you are to delete some code in a code listing, that code is struck through. In the following example, you are being instructed to delete the line defining variable `y` and to add a variable called `z`:

```
var x = "Python"
var y = "Java"
var z = "Kotlin"
```

Kotlin is a maturing language, and its coding conventions are still evolving over time. You will likely develop your own style, but we tend to adhere to JetBrains’ and Google’s Kotlin style guides:

- JetBrains’ coding conventions: kotlinlang.org/docs/coding-conventions.html
- Google’s style guide: developer.android.com/kotlin/style-guide

Using an eBook

If you are reading this book on an eReader, we want to point out that reading the code may be tricky at times. Longer lines of code may wrap to a second line, depending on your selected font size.

The longest lines of code in this book are 86 monospace characters, like this one.

```
println(playerCreateMessage(nameIsLong("Polarcubis, the Supreme Master of NyetHack")))
```

You can play with your eReader’s settings to find the best for viewing long code lines.

If you are reading on an iPad with Apple's Books app, we recommend you go to the Settings app, select Books, and set Full Justification OFF and Auto-hyphenation OFF.

When you get to the point where you are actually typing in code, we suggest opening the book on your PC or Mac in Adobe Digital Editions. (Adobe Digital Editions is a free eReader application you can download from www.adobe.com/products/digitaleditions.) Make the application window large enough so that you can see the code with no wrapping lines. You will also be able to see the figures in full detail.

Looking Forward

Take your time with the examples in this book. Once you get the hang of Kotlin's syntax, we think you will find the development process to be clear, pragmatic, and fluid. Until then, keep at it; learning a new language can be quite rewarding.

Part I

Getting Started

The first two chapters of this book walk you through the basics of using IntelliJ IDEA, the first-party IDE for Kotlin application development. You will get started by creating a simple project to get comfortable with the basic features of the language. To begin, you will explore Kotlin's types, which categorize the data that you work with in a program.

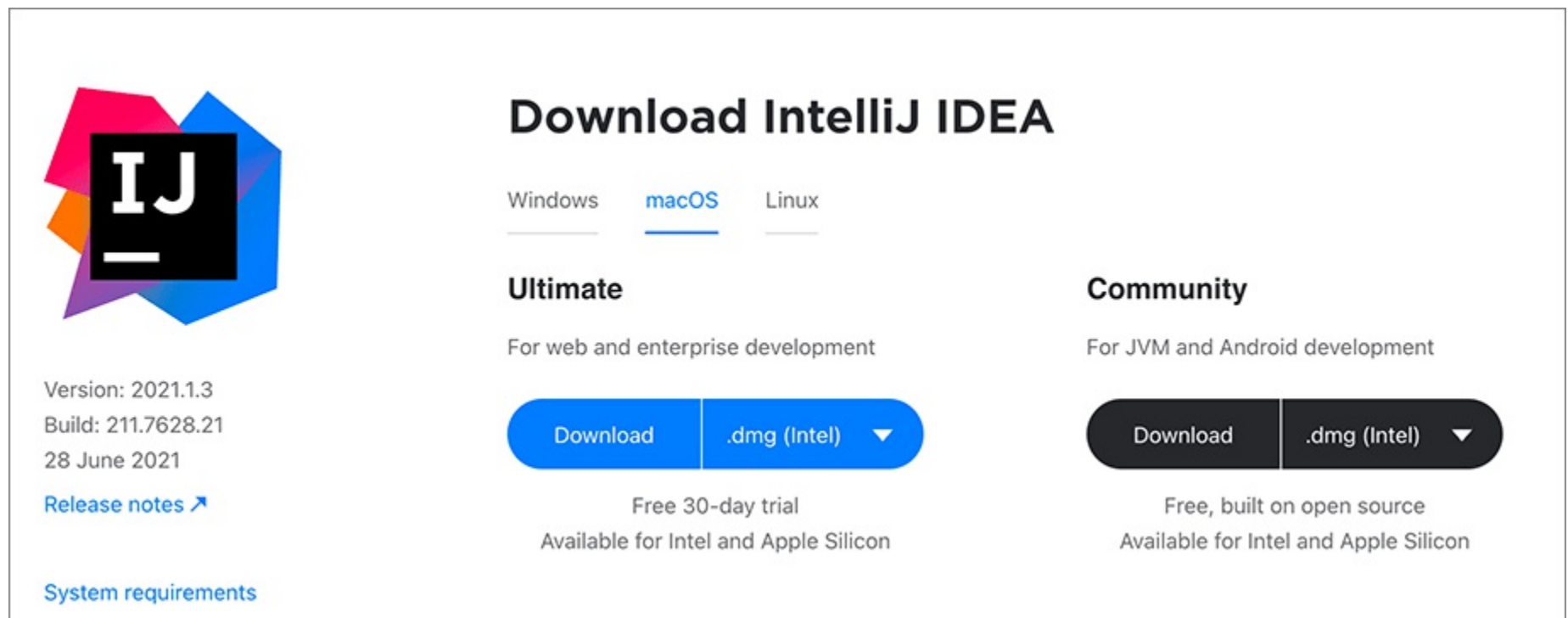
Your First Kotlin Application

In this chapter you will write your first Kotlin program, using IntelliJ IDEA. While completing this programming rite of passage, you will familiarize yourself with your development environment, create a new Kotlin project, write and run Kotlin code, and inspect the resulting output. The project you create in this chapter will serve as a testing bed for the fundamental language features you will use throughout your Kotlin applications.

Installing IntelliJ IDEA

IntelliJ IDEA is an integrated development environment (IDE) for Kotlin created by JetBrains (which also created the Kotlin language). To get started, download the IntelliJ IDEA Community Edition from the JetBrains website at jetbrains.com/idea/download (Figure 1.1).

Figure 1.1 Downloading IntelliJ IDEA Community Edition



When it has downloaded, follow the installation instructions for your platform as described on the JetBrains installation and set-up page at jetbrains.com/help/idea/installation-guide.html#standalone.

IntelliJ IDEA, called IntelliJ for short, helps you write well-formed Kotlin code. It also streamlines the development process with built-in tools for running, debugging, inspecting, and refactoring your code. You can read more about why we recommend IntelliJ for writing Kotlin code in [the section called *For the More Curious: Why Use IntelliJ?*](#) near the end of this chapter.

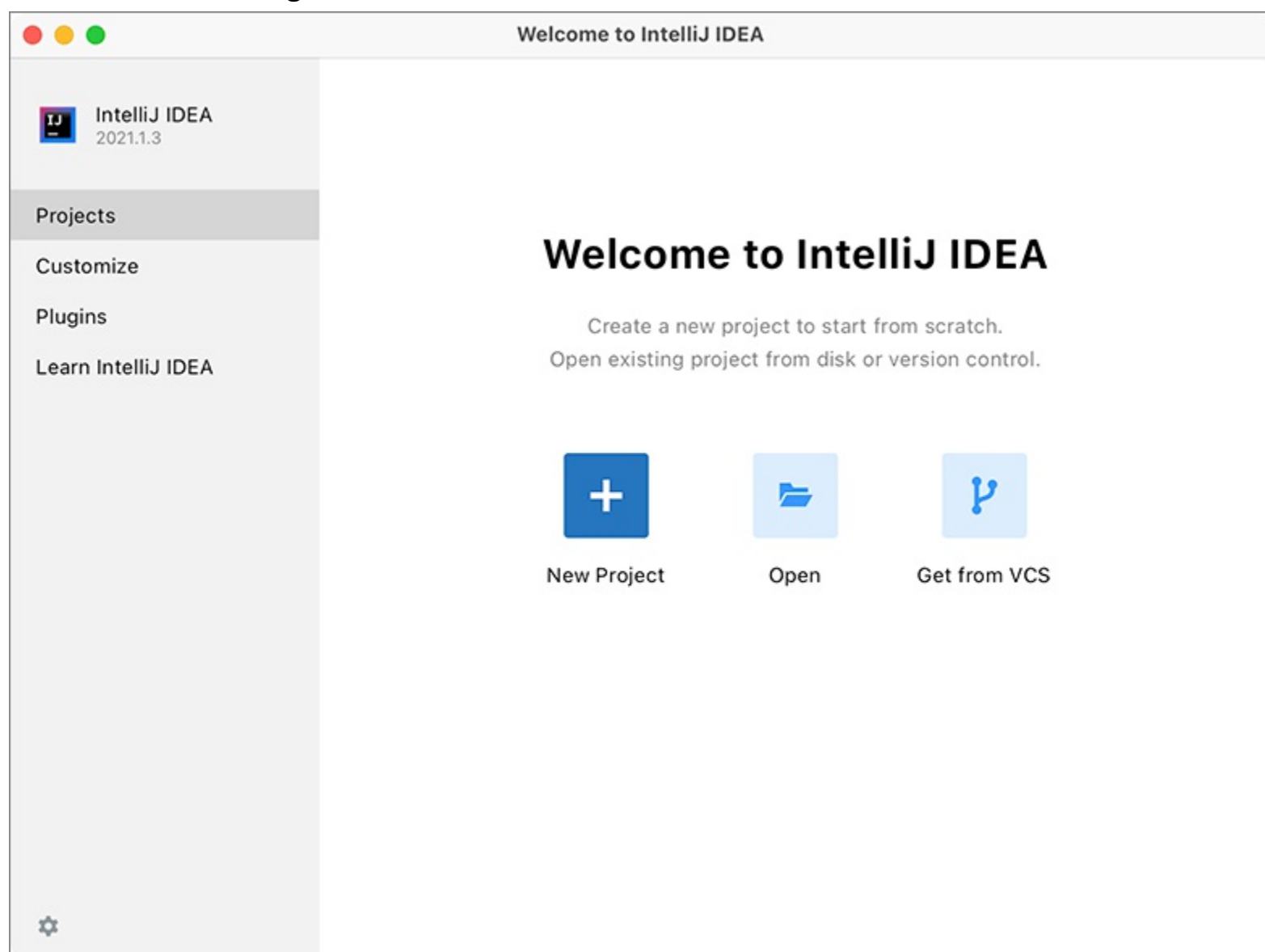
Your First Kotlin Project

Congratulations, you now have the Kotlin programming language and a powerful development environment to write it with. Now there is only one thing left to do: Learn to speak Kotlin fluently. First order of business – create a Kotlin project.

For much of this book, the projects you work on will focus on a fantasy game world in which a hero embarks on heroic quests, defeats villainous fiends, saves towns from certain peril, and generally does the things heroes do. Your first project builds a “bounty board,” a quest system that will guide the hero, named Madrigal, to tasks that require her attention. You will work on this project through [Chapter 7](#).

Open IntelliJ. You will be presented with the Welcome to IntelliJ IDEA dialog ([Figure 1.2](#)).

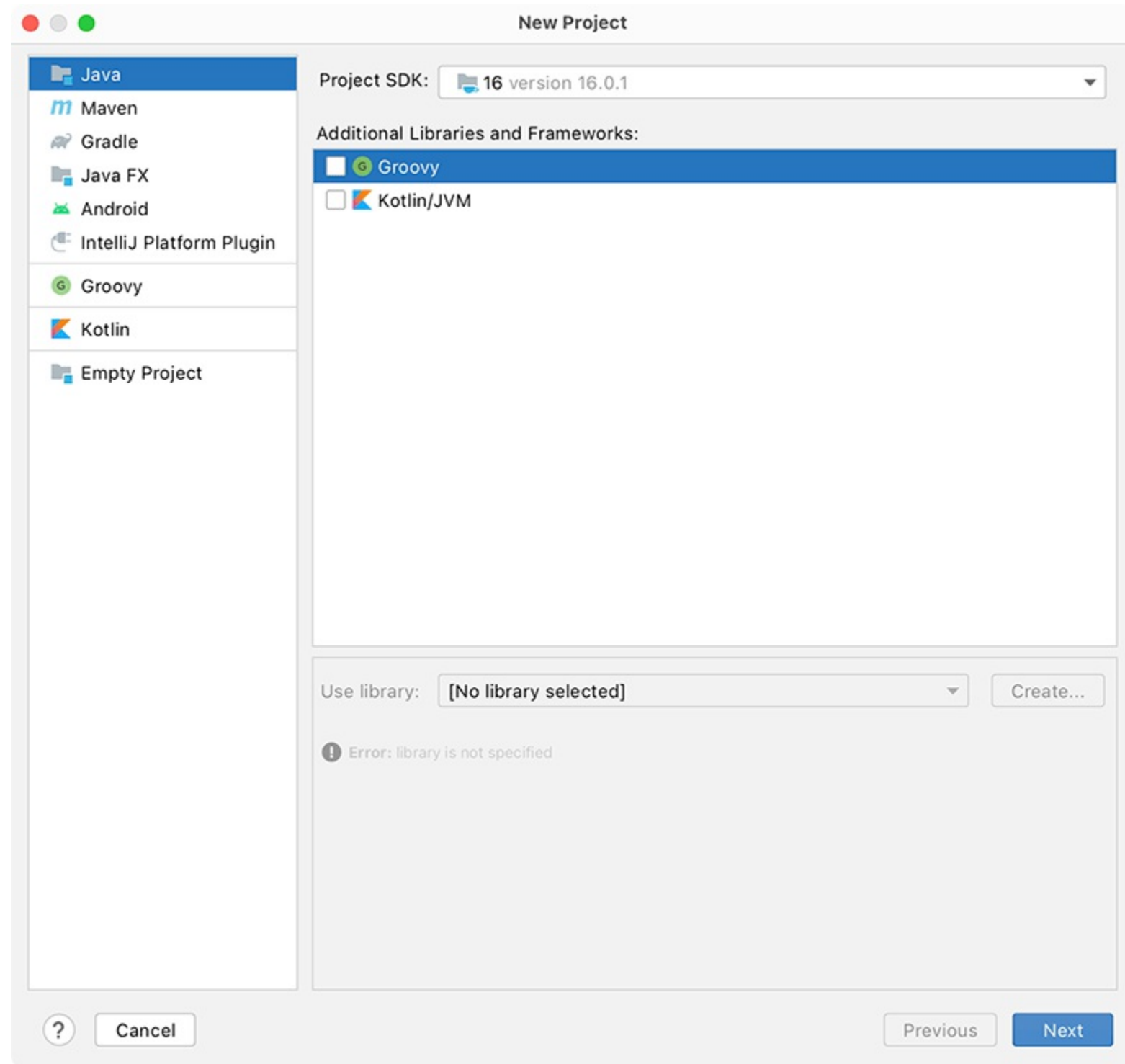
Figure 1.2 Welcome dialog



(If this is not the first time you have opened IntelliJ since installing it, you may be brought directly to the last project you had open. To get back to the Welcome screen, close the project using File → Close Project.)

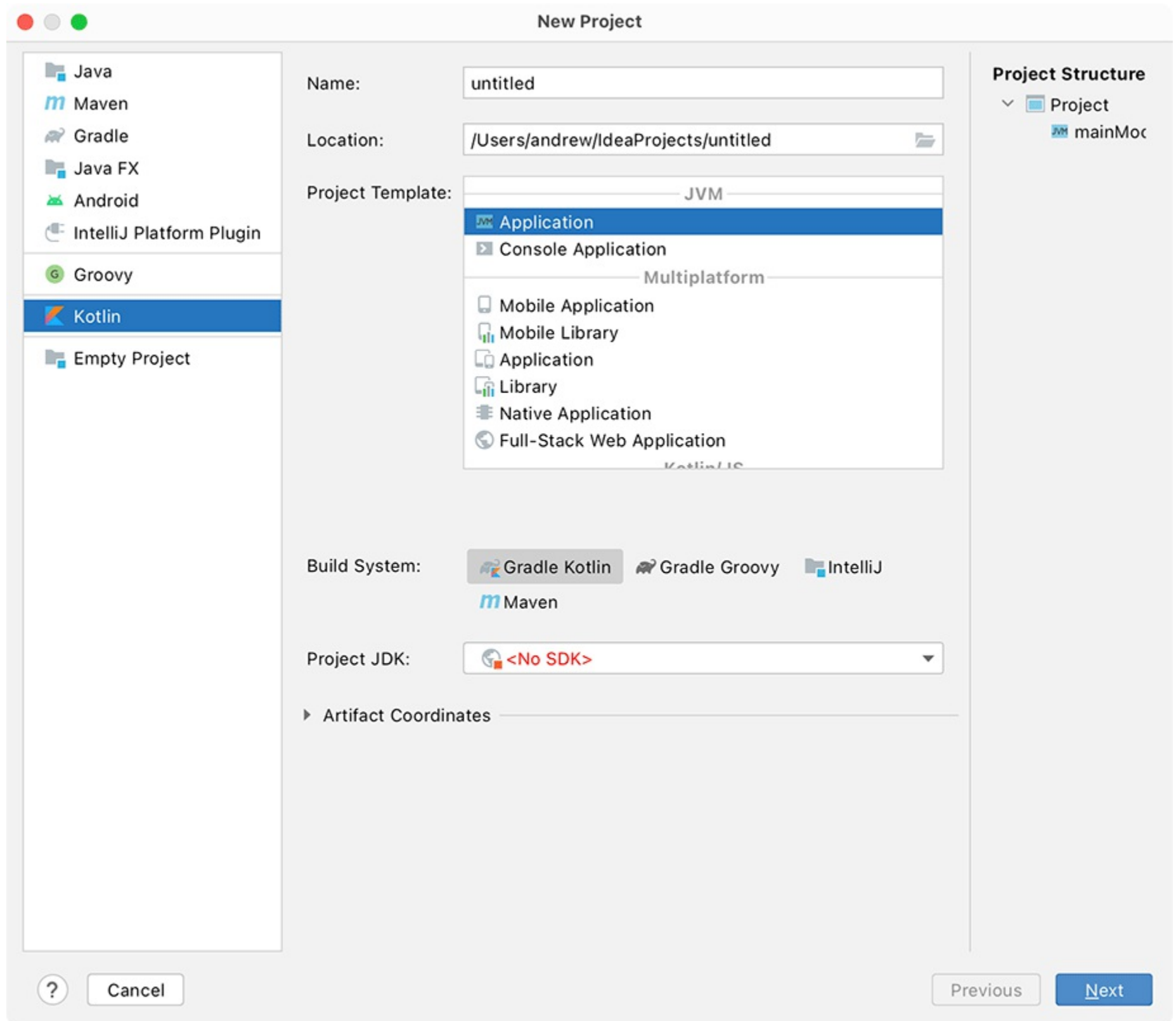
Click New Project. IntelliJ will display the New Project dialog, as shown in [Figure 1.3](#).

Figure 1.3 New Project dialog



In the New Project dialog, select Kotlin on the left, as shown in [Figure 1.4](#).

Figure 1.4 Creating a Kotlin project



You can use IntelliJ to write code in languages other than Kotlin, including Java and Groovy out of the box. With additional plugins, IntelliJ can also be used to write code in languages like Python, Scala, Dart, and Rust, to name a few. Selecting Kotlin from the language choices on the left side of the New Project dialog tells IntelliJ you intend to use Kotlin.

Now take a look at the project settings in the dialog’s center pane.

At the top of the New Project dialog, enter bounty-board for the Name. The Location field will autopopulate. You can leave the location as-is or select a new location by clicking the folder icon to the right of the field.

The Project Template menu has various options under three main headings: JVM, Multiplatform, and Kotlin/JS (as well as an Experimental section). Select Application under the JVM heading. This tells IntelliJ you intend to write Kotlin code that [targets](#), or runs on, the Java Virtual Machine.

Kotlin code can be compiled against any of the three targets that are supported by the language: the Java Virtual Machine, native x86 and ARM platforms (included in the Multiplatform section of the Project Template list), and JavaScript (abbreviated as JS). Because Kotlin can compile into any of these three targets, you will sometimes see the terms Kotlin/JVM, Kotlin/Native, and Kotlin/JS to specify the “flavor” of Kotlin based on the target.

Kotlin/JVM is what most people think of when you say “Kotlin.” You use Kotlin/JVM whenever you write code that targets the Java Virtual Machine. Similarly, Kotlin/JS is what you would use when you want to output JavaScript code from your Kotlin code. And Kotlin/Native is a catch-all for all Kotlin programs that compile into native machine code. As the Multiplatform heading suggests, you can use Kotlin/Native to build software for many platforms, such as iOS libraries, native desktop applications, and embedded devices if you are ambitious.

(From here on, we will refer to the Java Virtual Machine as “JVM,” as it is commonly called in the Java developer community. You can learn more about targeting the JVM in [the section called For the More Curious: Targeting the](#)

[JVM](#) near the end of this chapter.)

Kotlin/JVM is the most mature of these three platforms. It was the first supported by Kotlin and continues to be one of the most common platforms for Kotlin development. For the majority of this book, you will be using Kotlin/JVM. There are several reasons we have made this decision:

- The JVM makes it much easier to achieve platform independence: Anywhere the JVM runs, so can your code. You can distribute one binary that will work on every computer, regardless of architecture.
- Java is a very mature language and has many APIs that you will use throughout the book. These high-level APIs make it much easier to accomplish certain tasks compared to the lower-level APIs available in native platforms.
- The examples in this book will be executed inside your IDE, but can be run from any terminal. This makes JavaScript a less preferable target, because it primarily runs in a browser instead of a terminal.

Kotlin behaves very similarly across these three targets, so knowledge about Kotlin/JVM applies directly to Kotlin/JS and Kotlin/Native. Not all APIs are available across all targets, so we will note when we are using an API that can only be used on the JVM. You will learn more about how to use Kotlin on other platforms in [Chapter 24](#), [Chapter 25](#), and [Chapter 26](#).

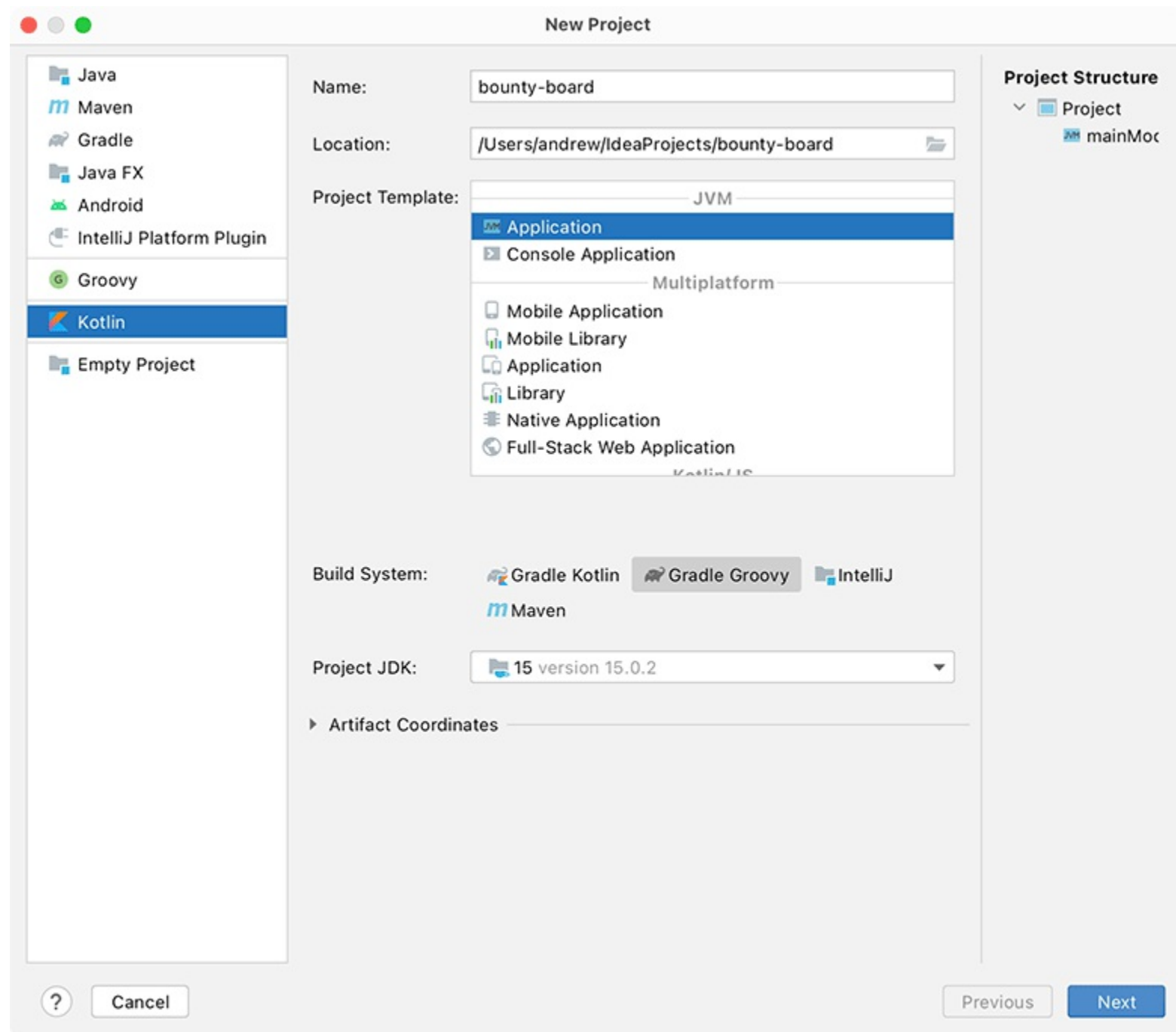
Back to setting up your new project. For the Build System, select Gradle Groovy. Finally, for the Project JDK, select a version of Java to link your project to the Java Development Kit (JDK). We recommend using a version between Java 8 (often listed as Java 1.8) and Java 15.

If you do not see an appropriate version of Java listed in the dropdown, IntelliJ did not detect an installation on your computer. If you know for a fact that you have a JDK installed and would like to use that version, you can use the Add JDK... option and locate the installation on disk. Otherwise, IntelliJ can install one for you via the Download JDK... option. (We recommend setting the Version to 15 and setting the Vendor to AdoptOpenJDK (HotSpot), but any vendor should work.) Once IntelliJ finishes the installation, you are good to go.

Why do you need the JDK to write a Kotlin program? The JDK gives IntelliJ access to the JVM and to Java tools that are necessary for converting your Kotlin code to bytecode (more on that in a moment). Technically, any version 6 or greater will work. But our experience, as of this writing, is that JDK 8 or newer works most seamlessly.

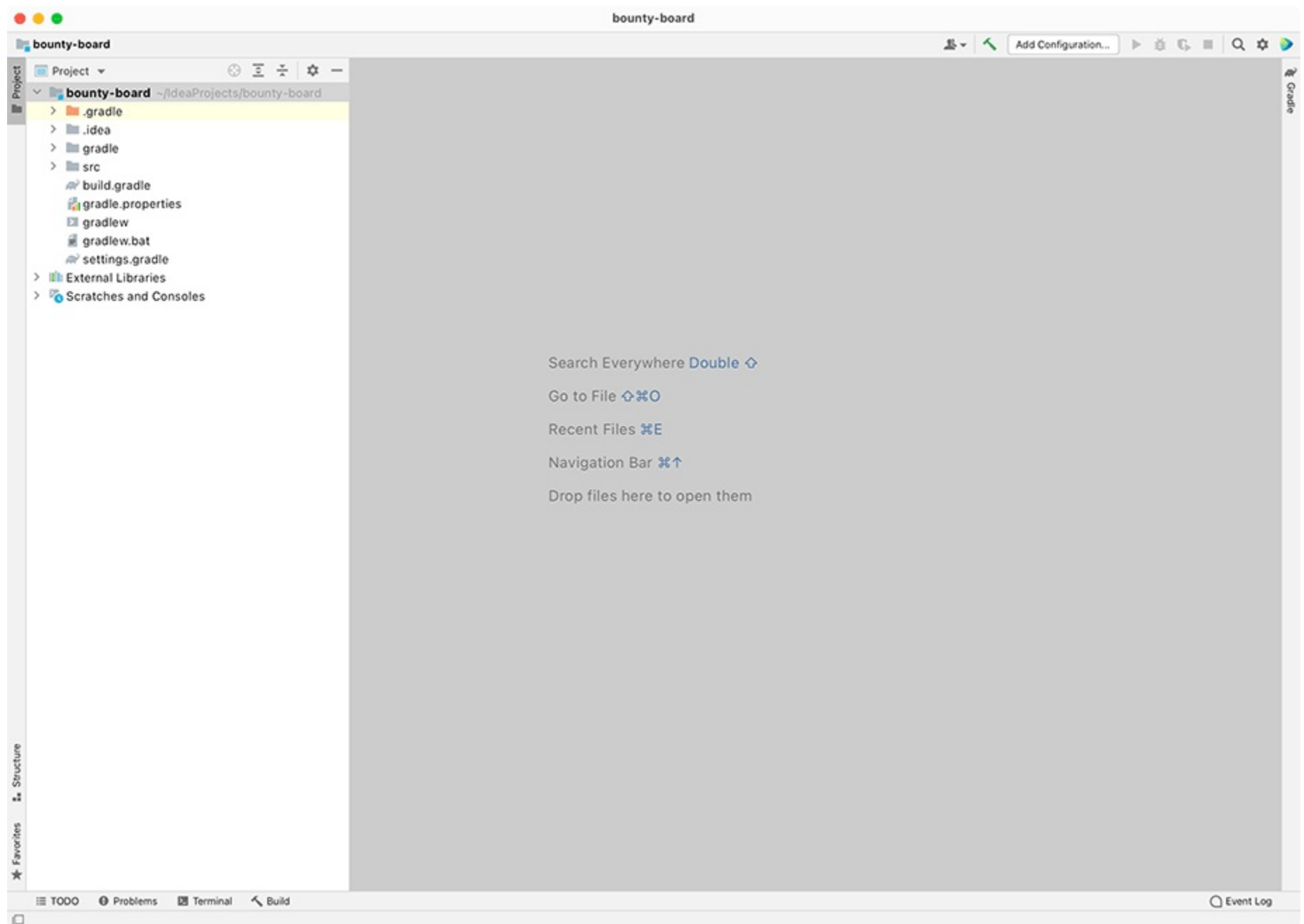
When your settings dialog looks like [Figure 1.5](#), click Next, then click Finish on the next dialog to confirm the settings there.

Figure 1.5 Setting up a project



IntelliJ will generate a project named bounty-board and display the new project in a default two-pane view ([Figure 1.6](#)). On disk, IntelliJ creates a folder and a set of subfolders and project files in the location specified in the Project location field.

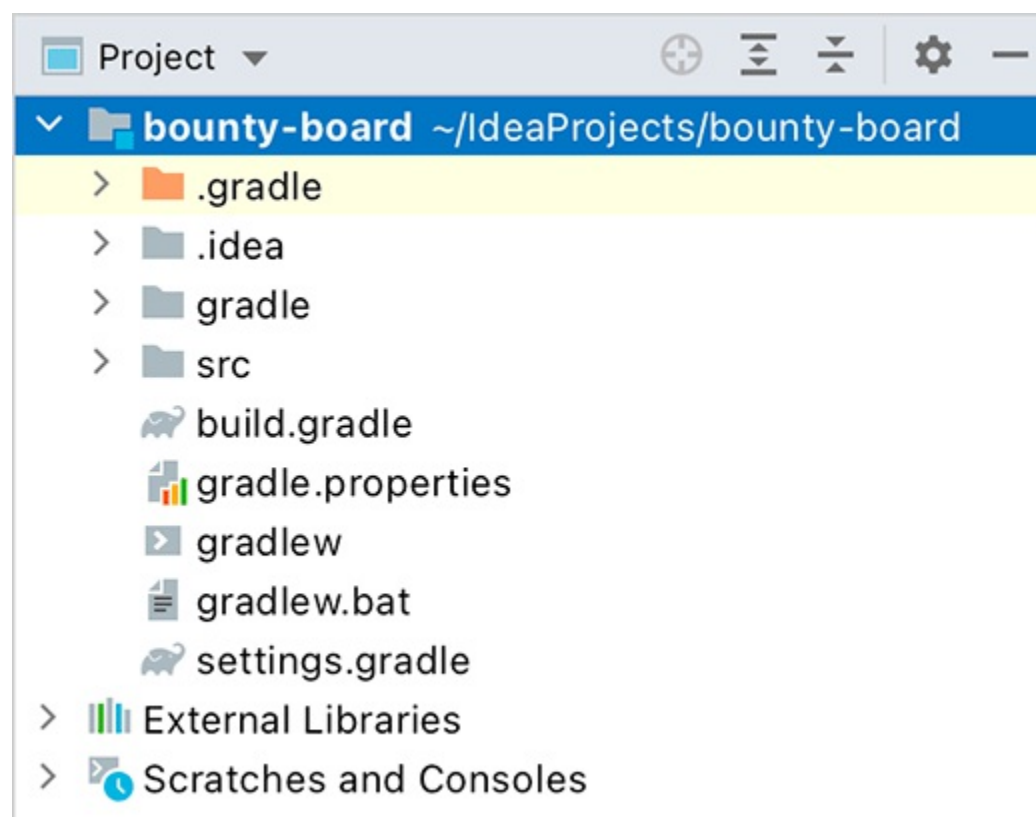
Figure 1.6 Default two-pane view



The pane on the left shows the [project tool window](#). The pane on the right is currently empty. This is where you will view and edit the contents of your Kotlin files in the [editor](#).

The project tool window displays the files contained in the bounty-board project, as shown in [Figure 1.7](#).

Figure 1.7 Project view



A [project](#) includes all the source code for your program, along with information about dependencies and configurations. A project can be broken down into one or more [modules](#), which are like subprojects. By default, a new project has one module, which is all you need for your simple first project.

The `.gradle` and `.idea` folders shown in [Figure 1.7](#) may be hidden when you open the project in a file explorer. They are “behind the scenes” folders that you will not need to access; `.gradle` contains caches used by the build

system, and `.idea` contains settings files for the project and your IDE.

The `gradle` folder and files like `gradlew` and `gradlew.bat` are for the *Gradle wrapper*, a self-contained copy of the Gradle build system that you can use without having to install Gradle to your computer. The `build.gradle`, `gradle.properties`, and `settings.gradle` files are configuration files for the Gradle build system. They define various pieces of information such as the project name, language level, project modules, and dependencies in your project. Leave these autogenerated files as they are.

The External Libraries entry contains information about libraries the project depends on. If you expand this entry, you will see that IntelliJ automatically added Java and several Kotlin standard library packages as dependencies for your project.

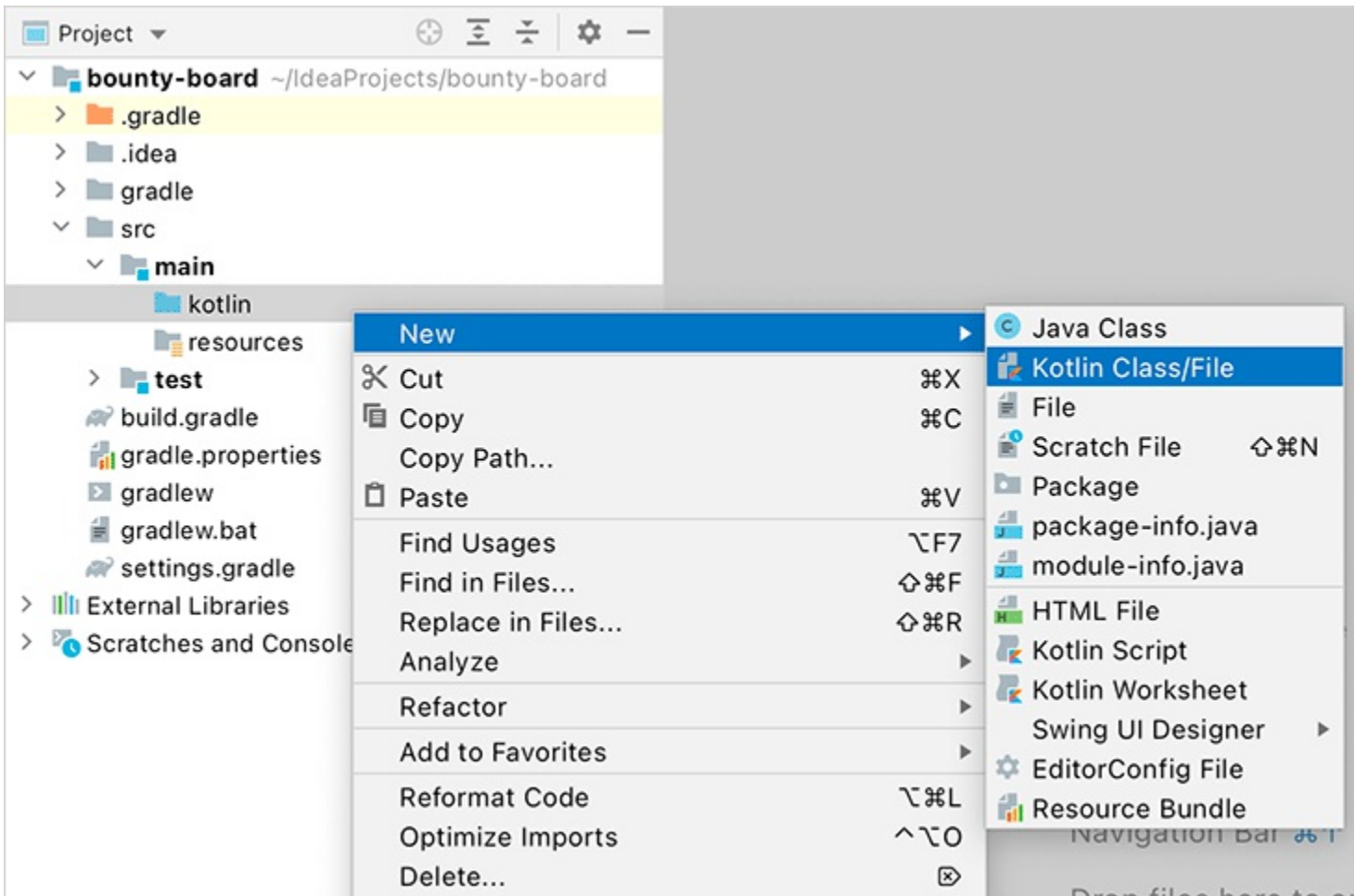
(You can learn more about IntelliJ project structure on the JetBrains documentation website at jetbrains.org/intellij/sdk/docs/basics/project_structure.html. You can also learn more about the structure of a Gradle project on Gradle’s website at docs.gradle.org/current/userguide/organizing_gradle_projects.html.)

The `src/main/kotlin` folder is where you will place all the Kotlin files you create for your project. And with that, it is time to create and edit your first Kotlin file.

Creating your first Kotlin file

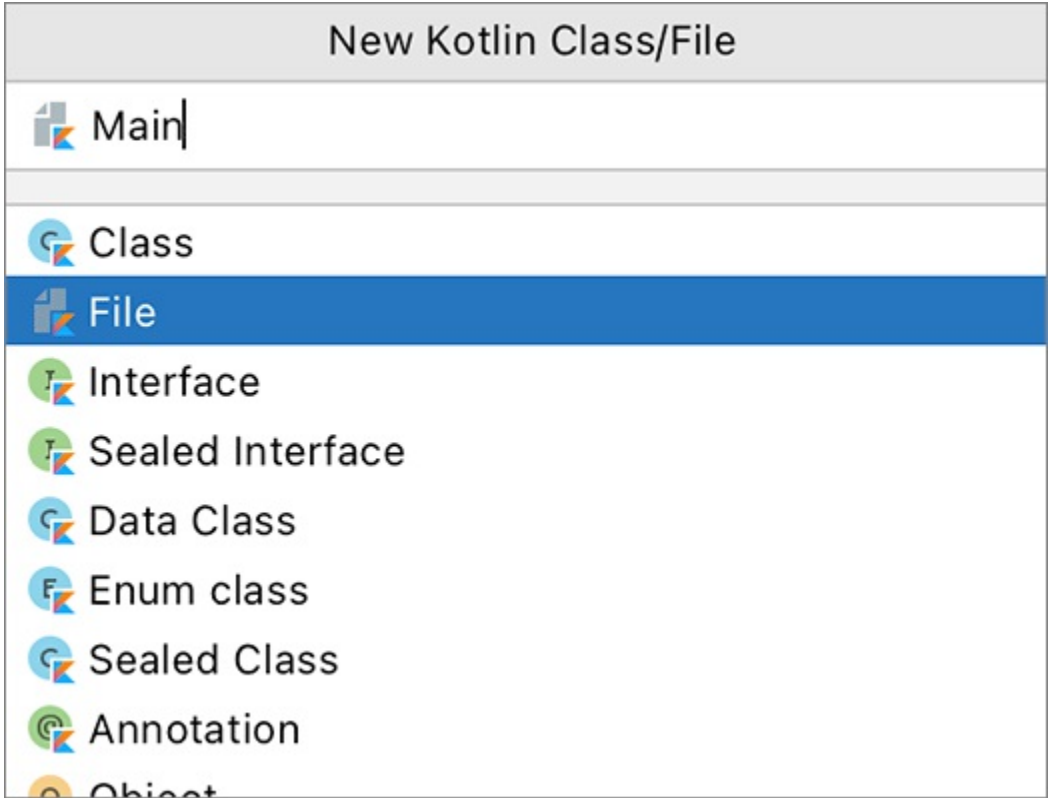
Disclose and right-click the `src/main/kotlin` folder in the project tool window. Select New and then Kotlin Class/File from the menu that appears (Figure 1.8).

Figure 1.8 Creating a new Kotlin file



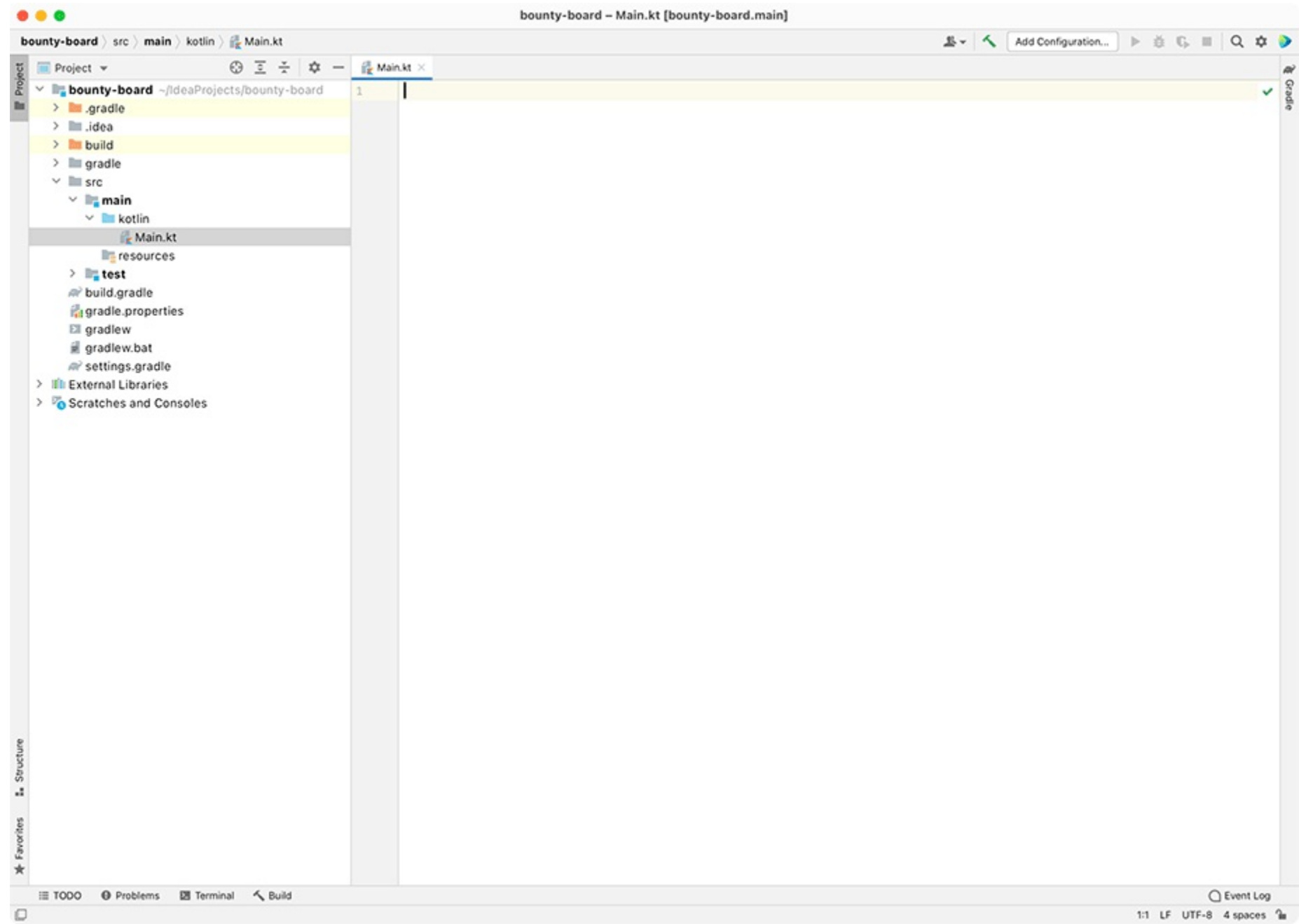
In the New Kotlin Class/File dialog, type `Main` in the Name field and double-click `File` in the list of file types below (Figure 1.9).

Figure 1.9 Naming the file



IntelliJ will create a new file in your project, `src/main/kotlin/Main.kt`, and display the contents of the file in the editor on the righthand side of the IntelliJ window ([Figure 1.10](#)). The `.kt` extension indicates that the file contains Kotlin, just like the `.java` extension is used for Java files and `.py` for Python files.

Figure 1.10 Empty `Main.kt` file displays in editor



At last, you are ready to write Kotlin code. Give your fingers a little stretch and go for it. Type the following code into the `Main.kt` editor. (Remember that throughout this book, code you are to enter is shown in bold.)

Listing 1.1 “Hello, world!” in Kotlin (`Main.kt`)

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

The code you just wrote might look unfamiliar. Do not fear – by the end of this book, reading and writing Kotlin will feel like second nature. For now, it is enough to understand the code at a high level.

The code in [Listing 1.1](#) defines a new [function](#). A function is a group of instructions that can be run later. You will learn in great detail how to define and work with functions in [Chapter 4](#).

This particular function – the **main** function – has a special meaning in Kotlin. The **main** function indicates the starting place for your program. This is called the [application entry point](#), and one such entry point must be defined for bounty-board (or any program) to be runnable. Every project you write in this book will start with a **main** function.

Your **main** function contains one instruction (also known as a [statement](#)): `println("Hello, world!")`. **println()** is also a function, one that is built into the [Kotlin standard library](#) and is available on all targets that Kotlin supports. When the program runs and `println("Hello, world!")` is executed, IntelliJ will print the contents of the parentheses (without the quotation marks, so in this case `Hello, world!`) to the screen.

Running your Kotlin file



Shortly after you finish typing the code in [Listing 1.1](#), IntelliJ will display a green , known as the “run button,” to the left of the first line ([Figure 1.11](#)). (If the icon does not appear, or if you see a red line underneath the filename in the tab or under any of the code you entered, this means you have an error in your code. Double-check that you typed the code exactly as shown in [Listing 1.1](#).)

Figure 1.11 Run button



It is time for your program to come to life and greet the world. Click the run button. Select Run 'MainKt' from the menu that appears ([Figure 1.12](#)). This tells IntelliJ you want to see your program in action.

Figure 1.12 Running Main.kt



When you run your program, IntelliJ executes the code inside the curly braces (`{}`), one line at a time, and then terminates execution. It also displays a new tool window at the bottom of the IntelliJ window ([Figure 1.13](#)).

Figure 1.13 Run tool window (console)



This window is the [run tool window](#), also known as the [console](#) (which is what we will call it from now on). It displays information about what happened as IntelliJ executed your program, as well as any output your program prints. You should see `Hello, world!` printed in your console. You should also see `Process finished with exit code 0`, indicating successful completion. This line appears at the end of all console output when there is no error; we will not show it in console results from now on.

Compilation and execution of Kotlin/JVM code

A lot goes on in the short time between when you select the run button's `Run 'MainKt'` option and when you see `Hello, world!` print to the console.

First, IntelliJ [compiles](#) the Kotlin code using the `kotlinc-jvm` [compiler](#). A compiler is a program that translates source code into a lower-level language to create an executable program. If you were targeting a different platform, IntelliJ would use `kotlinc-js` or `kotlinc-native`, as appropriate.

All three variations of `kotlinc` work in roughly the same way. When `kotlinc-jvm` runs, it translates the Kotlin code you wrote into [bytecode](#), the language the JVM “speaks.” If `kotlinc` has any problems translating your Kotlin code, it will display an error message (or messages) giving you a hint about how to fix the issues. Otherwise, if the compilation process goes smoothly, IntelliJ moves on to the execution phase.

In the execution phase, the bytecode that was generated by `kotlinc-jvm` is executed on the JVM. The console displays any output from your program, such as printing the text you specified in your call to the `println()` function, as the JVM executes the instructions.

When there are no more bytecode instructions to execute, the JVM terminates. IntelliJ shows the termination status in the console, letting you know whether execution finished successfully or with an error code.

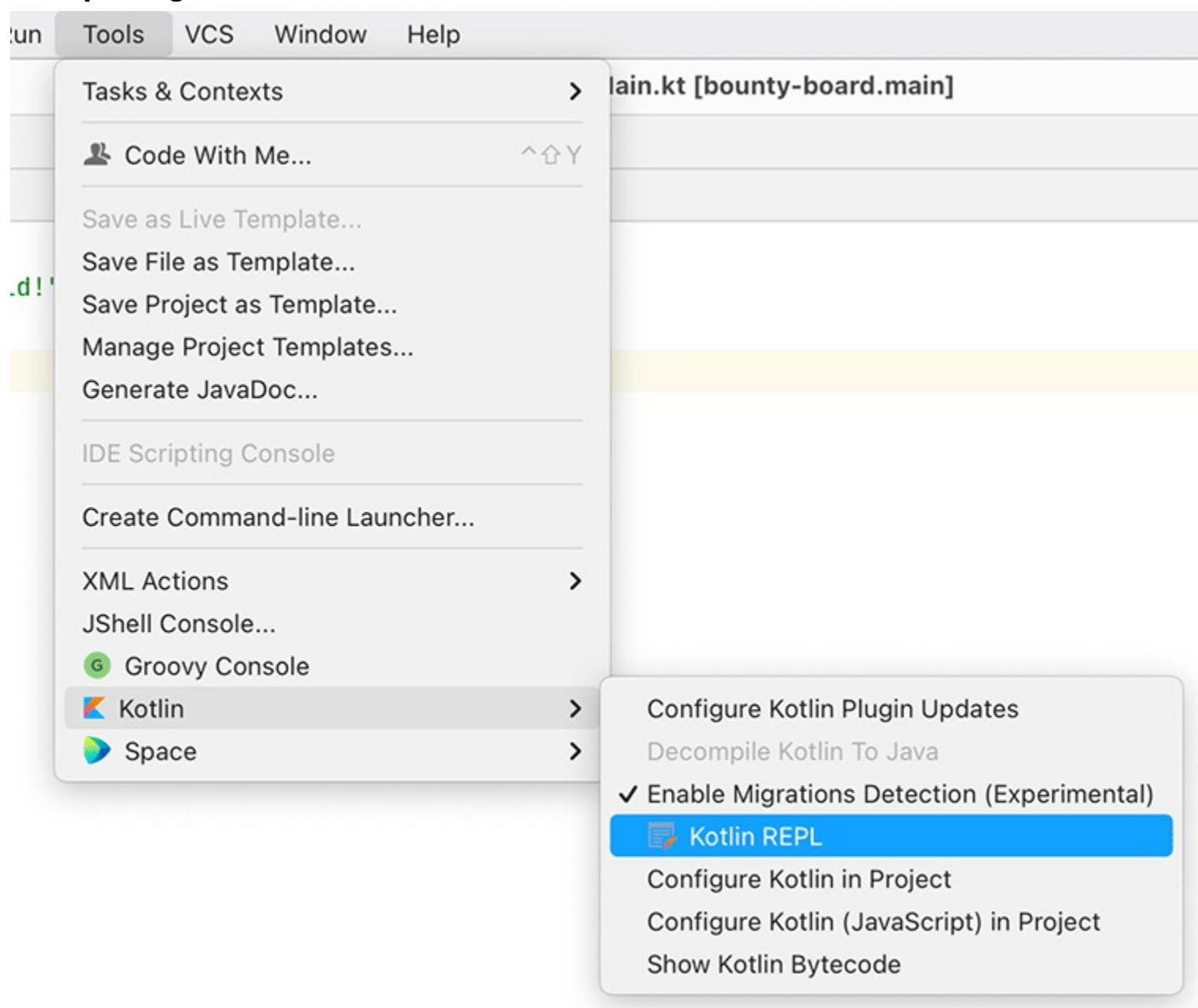
You will not need a comprehensive understanding of the Kotlin compilation process to work through this book. We will, however, discuss bytecode in more detail in [Chapter 2](#).

The Kotlin REPL

Sometimes you might want to test out a small bit of Kotlin code to see what happens when you run it, similar to how you might use a piece of scratch paper to jot down steps for a small calculation. This is especially helpful as you are learning the Kotlin language. Luckily for you, IntelliJ provides a tool for quickly testing code without having to create a file. This tool is called the [Kotlin REPL](#). We will explain the name in a moment – for now, open it up and see what it can do.

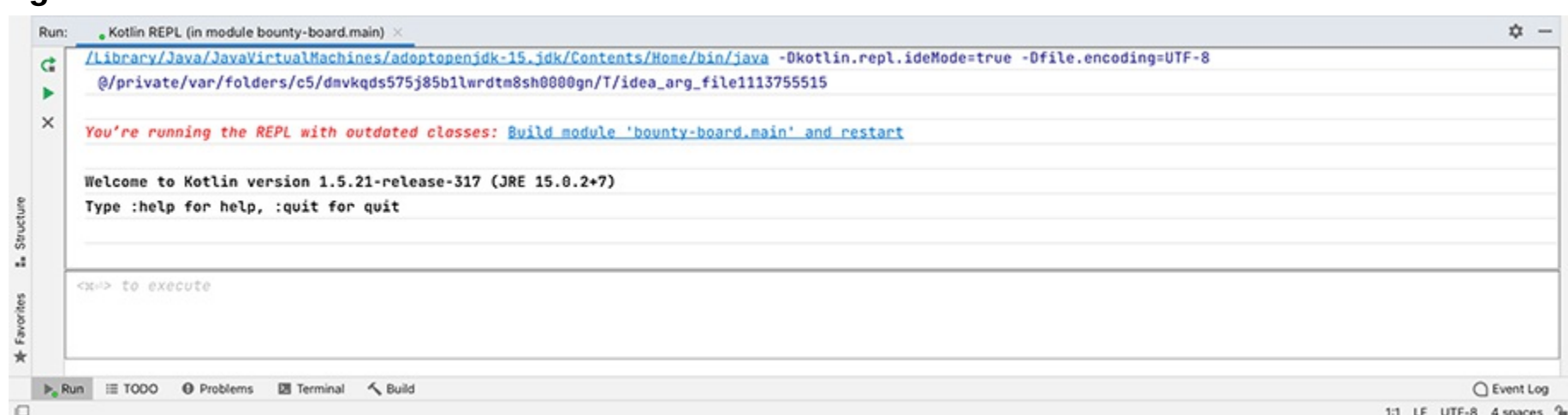
In IntelliJ, open the Kotlin REPL tool window by selecting Tools → Kotlin → Kotlin REPL ([Figure 1.14](#)).

Figure 1.14 Opening the Kotlin REPL tool window



IntelliJ will display the REPL at the bottom of the window ([Figure 1.15](#)).

Figure 1.15 The Kotlin REPL tool window



You can type code into the REPL, just like in the editor. The difference is that you can have it evaluated quickly, without compiling an entire project.

(You may see red warning text when starting the REPL or warnings about “running the REPL with outdated classes.” You can generally ignore these kinds of warnings. Generic issues about the platform or JVM will do no harm when using the REPL, and because you will not be accessing code from bounty-board in the REPL, you do

not have to worry about the outdated classes warning.)

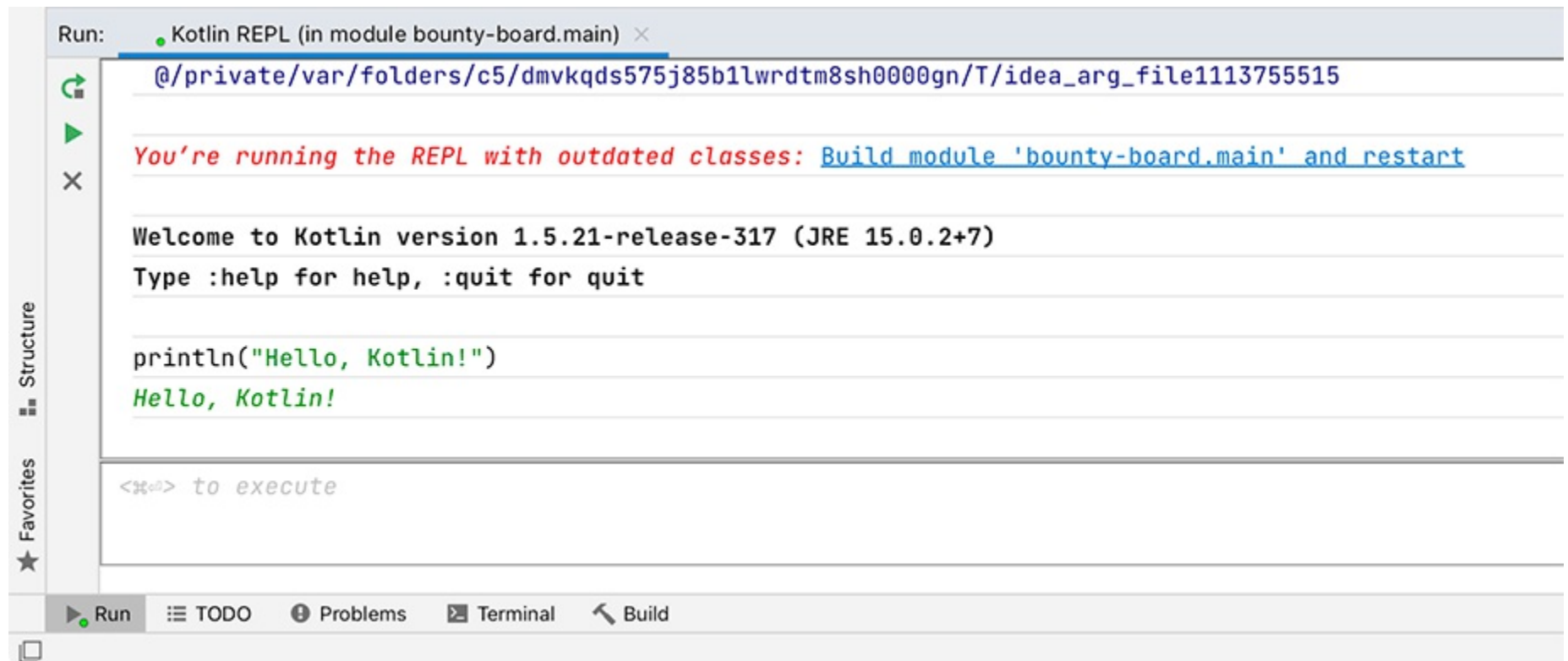
Enter the following code in the REPL:

Listing 1.2 “Hello, Kotlin!” (REPL)

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

After you enter the text, press Command-Return (Ctrl-Enter) to evaluate the code in the REPL. After a moment, you will see the resulting output underneath, which should read `Hello, Kotlin!` ([Figure 1.16](#)).

Figure 1.16 Evaluating the code



REPL is short for “read, evaluate, print, loop.” You type in a piece of code at the prompt and submit it by clicking the green run button on the REPL’s left side or by pressing Command-Return (Ctrl-Enter). The REPL then *reads* the code, *evaluates* (runs) the code, and *prints* out the resulting value or side effect. Once the REPL finishes executing, it returns control to you and the process *loop* starts all over.

Your Kotlin journey has begun! You accomplished a great deal in this chapter, laying the foundation for your growing knowledge of Kotlin programming. In the next chapter, you will begin to dig into the language’s details by learning about how you can use variables, constants, and types to represent data.

For the More Curious: Why Use IntelliJ?

Kotlin can be written using any plain text editor. However, we recommend using IntelliJ, especially as you are learning. Just as text editing software that offers spell check and grammar check makes writing a well-formed prose essay easier, IntelliJ makes writing well-formed Kotlin easier. IntelliJ helps you:

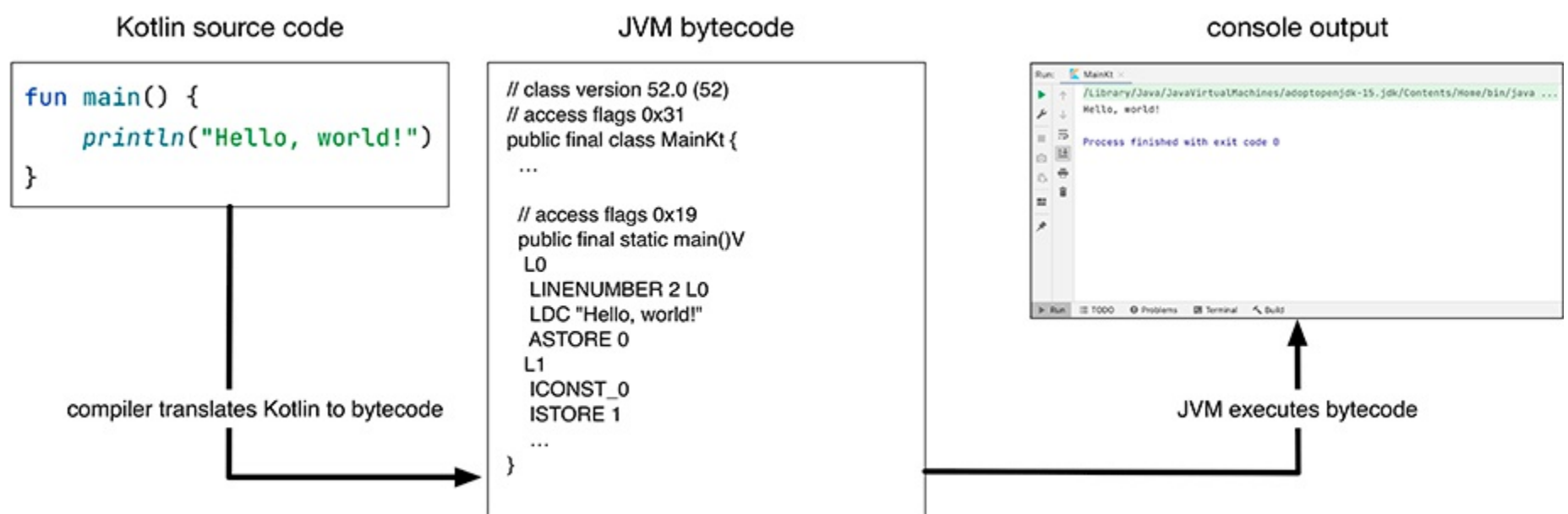
- write syntactically and semantically correct code with features like syntax highlighting, context-sensitive suggestions, and automatic code completion
- run and debug your code with features like debug breakpoints and real-time code stepping when your application is running
- restructure existing code with refactoring shortcuts (like rename and extract constant) and code formatting to clean up indentation and spacing

Also, because Kotlin was created by JetBrains, the integration between IntelliJ and Kotlin is carefully designed – often leading to a delightful editing experience. As an added bonus, IntelliJ is the basis of Android Studio, so shortcuts and tools you learn here will translate to using Android Studio, if that is your thing.

For the More Curious: Targeting the JVM

The JVM is a piece of software that knows how to execute a set of instructions, called bytecode. “Targeting the JVM” means compiling, or translating, your Kotlin source code into Java bytecode, with the intention of running that bytecode on the JVM ([Figure 1.17](#)).

Figure 1.17 Compilation and execution flow



Each platform, such as Windows or macOS, has its own instruction set. The JVM acts as a bridge between the bytecode and the different hardware and software environments the JVM runs on, reading a piece of bytecode and calling the corresponding platform-specific instruction(s) that map to that bytecode. Therefore, there are different versions of the JVM for different platforms. This is what allows Kotlin developers to write platform-independent code that can be written one time and then compiled into bytecode and executed on different devices regardless of their operating systems.

Since Kotlin can be converted to bytecode that the JVM can execute, it is considered a JVM language. Java is perhaps the most well-known JVM language, because it was the first. However, other JVM languages, such as Scala, Groovy, and Kotlin, have emerged to address some shortcomings of Java from the developer perspective.

Challenge: REPL Arithmetic

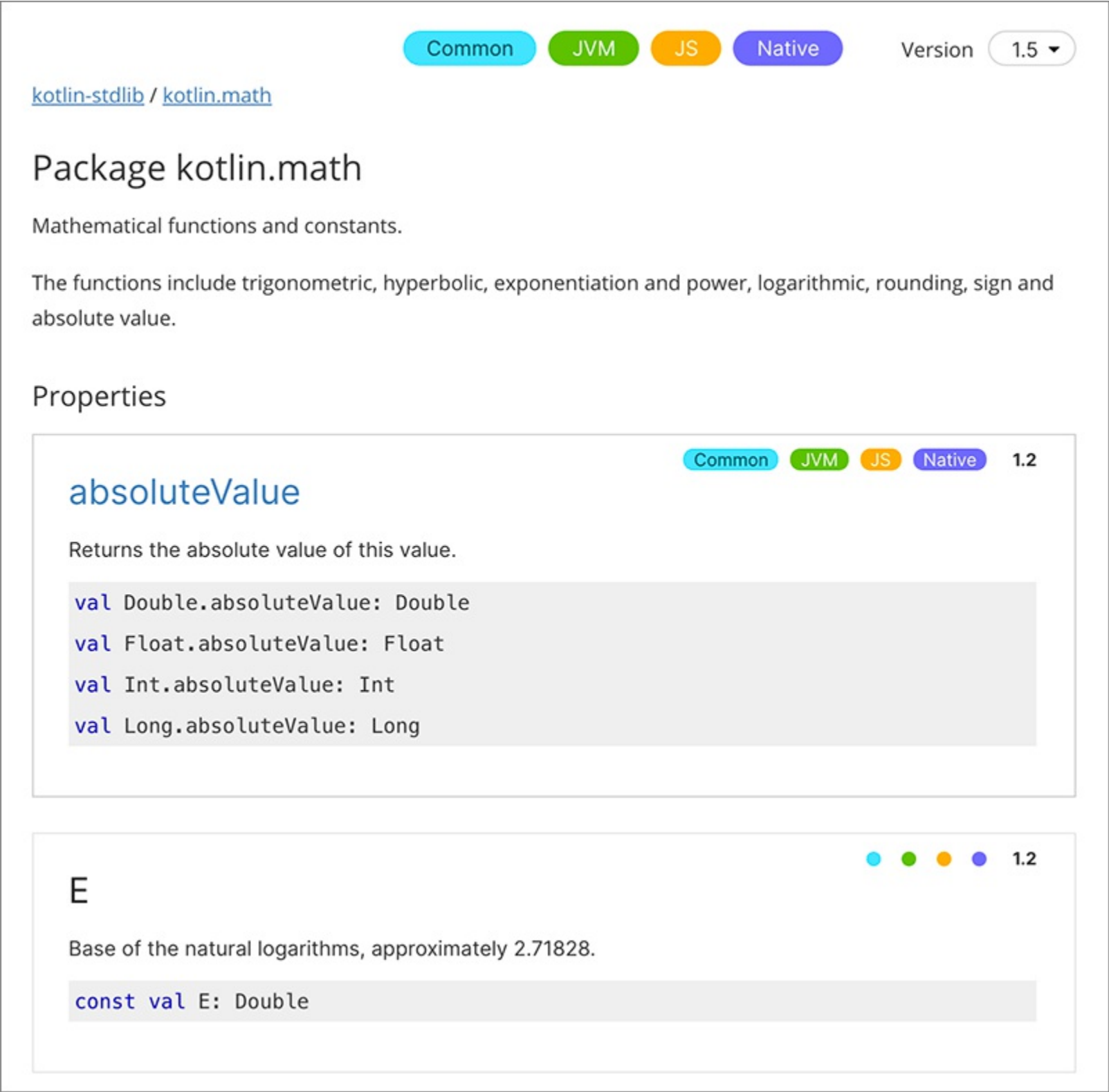
Many of the chapters in this book end with one or more challenges. The challenges are for you to work through on your own to deepen your understanding of Kotlin and get a little extra experience.

Use the REPL to explore how arithmetic [operators](#) in Kotlin work: +, -, *, /, and %. For example, type (9+12)*2 into the REPL. Does the output match what you expected?

If you want to dive deeper, look over the mathematical functions available in the Kotlin standard library at kotlinlang.org/api/latest/jvm/stdlib/kotlin.math/ and try them out in the REPL. For example, try min(94, -99), which will tell you the minimum of the two numbers provided in parentheses.

You will want to get comfortable with the kotlinlang.org website and especially the language documentation there. If you are planning to use Kotlin outside the JVM, pay attention to the colored circles when you read the API references. These indicate which platforms a given API supports. Many APIs, like absoluteValue (shown in [Figure 1.18](#)) are considered common APIs and work on every target that Kotlin supports.

Figure 1.18 Platform support in the Kotlin API documentation



Variables, Constants, and Types

This chapter will introduce you to variables, constants, and Kotlin's basic data types – fundamental elements of any program. You use [variables](#) and [constants](#) to store values and pass data around in your application. [Types](#) describe the particular kind of data that is held by a constant or variable.

There are important differences between each of the data types and between variables and constants that shape how they are used.

Types

Variables and constants have a data type that you specify. The type describes the data that is held by a variable or constant and tells the compiler how [type checking](#) – a feature in Kotlin that prevents the assignment of the wrong kind of data to a variable or constant – will be handled.

To see this idea in action, you are going to edit the `Main.kt` file in the bounty-board project you created in [Chapter 1](#). If you closed IntelliJ since the last chapter, launch it again. The bounty-board project will likely open automatically, because IntelliJ reopens your most recent project. If it does not, you can open bounty-board from the list of recent files in the center of the welcome dialog or by selecting `File` → `Open Recent` → `bounty-board`.

Declaring a Variable

Imagine you are writing an adventure game that assigns players quests to embark on. These quests should scale in difficulty as the player grows stronger and progresses through the game. You will likely want a variable for keeping track of the player's game progress.

In `Main.kt`, create your first variable, called `playerLevel`, and assign it a value:

Listing 2.1 Declaring a `playerLevel` variable (`Main.kt`)

```
fun main() {  
    println("Hello, world!")  
    var playerLevel: Int = 4  
    println(playerLevel)  
}
```

Here, you have assigned an instance of the type **Int** to a variable called `playerLevel`. Let's walk through each part of the code.

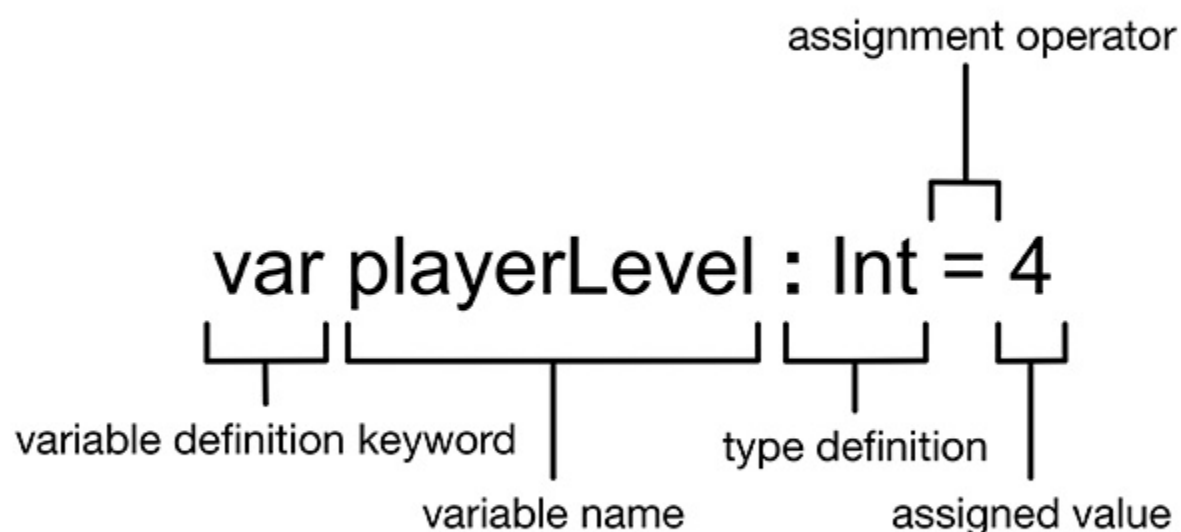
You defined a variable using the keyword `var`, which indicates that you want to declare a new variable, followed by the new variable's name.

Next, you specified the type definition for the variable, `: Int`, which indicates that `playerLevel` will hold an integer (whole number) value.

Last, you used the [assignment operator](#) (`=`) to assign what is on the righthand side (an instance of the **Int** type, specifically 4) to what is on the lefthand side (`playerLevel`).

[Figure 2.1](#) shows the `playerLevel` variable's definition in diagram form.

Figure 2.1 Anatomy of a variable definition



After defining the variable, you print its value to the console using the **println** function.

Run the program by clicking the run button next to the **main** function and selecting Run 'MainKt'. You can also run by clicking the run button in the IntelliJ menu bar. The result printed to the console is 4, the value you assigned to `playerLevel`.

Now, try assigning `playerLevel` the value "thirty-two" instead. (The strike-through indicates code you are to delete.)

Listing 2.2 Assigning "thirty-two" to `playerLevel` (`Main.kt`)

```
fun main() {  
    println("Hello, world!")  
    var playerLevel: Int = 4  
    var playerLevel: Int = "thirty-two"  
    println(playerLevel)  
}
```

Run **main** again by clicking the run button. This time, the Kotlin compiler displays an error:

```
e: Main.kt: (3, 28): Type mismatch: inferred type is String but Int was expected
```

When you typed this code, you may have noticed the red underline beneath "thirty-two". This is IntelliJ's signal that the program has an error. Hover over "thirty-two" to read the details of the detected problem

([Figure 2.2](#)).

Figure 2.2 Type mismatch disclosure tooltip

```
fun main() {  
    println("Hello, world!")  
    var playerLevel: Int = "thirty-two"  
    println(playerLevel)  
}
```

Type mismatch.
Required: Int
Found: String

[Change type of 'playerLevel' to 'String'](#)

[More actions...](#)

Kotlin uses a [static type system](#) – meaning the compiler labels the source code you define with types so it can ensure the code you write is valid. Static typing also means that once you define a variable, you cannot change its type after it has been declared.

IntelliJ checks code as you type it and notices compiler errors such as the one caused by incorrectly assigning to a variable with a value of a different type. This feature is called [static type checking](#), and it tells you about programming mistakes before you even compile the program.

To fix the error, assign `playerLevel` an **Int** value that matches its declared type by changing "thirty-two" back to 4:

Listing 2.3 Fixing the type error (`Main.kt`)

```
fun main() {  
    println("Hello, world!")  
    var playerLevel: Int = "thirty-two"  
    var playerLevel: Int = 4  
    println(playerLevel)  
}
```

As long as you respect the type system, a variable can be reassigned in the course of your program. If the player increases in rank, for example, you can assign a new value to the `playerLevel` variable. Set the `playerLevel` variable to 5, as shown:

Listing 2.4 Increasing `playerLevel` by resetting the value (`Main.kt`)

```
fun main() {  
    println("Hello, world!")  
    var playerLevel: Int = 4  
    println(playerLevel)  
  
    println("The hero embarks on her journey to locate the enchanted sword.")  
    playerLevel = 5  
    println(playerLevel)  
}
```

Run your updated **main** function to see this assignment in action. You will see the number 5 printed on its own line after the string The hero embarks on her journey to locate the enchanted sword..

Using the equals (=) operator, you can assign variables to take on new values. This works, but it would be better to increment `playerLevel` by 1 instead of setting it to 5. Incrementing the level is a better choice because it will make your code more flexible if, for example, you decide to change the player's starting level.

Update the reassignment to increment the variable instead. While you are making changes, update your program's greeting, which is beginning to feel out of place.

Listing 2.5 Incrementing `playerLevel` (`Main.kt`)

```
fun main() {  
    println("Hello, world!")  
    println("The hero announces her presence to the world.")  
    var playerLevel: Int = 4  
    println(playerLevel)  
  
    println("The hero embarks on her journey to locate the enchanted sword.")  
    playerLevel = 5  
    playerLevel += 1  
    println(playerLevel)  
}
```

After assigning the `playerLevel` variable a value of 4, you use the [addition and assignment operator](#) (+=) to add 1 to the original value. Run the program again. You will see the new greeting, and the player level will increase from 4 to 5 as before.

Kotlin offers other ways to assign values as well. Because you are incrementing `playerLevel` by 1, you could use the [increment operator](#) (++) instead of the += operator, like this:

```
playerLevel++
```

To subtract instead of add, you could use the [decrement operator](#) (--) to subtract 1 or the [subtraction and](#)

[*assignment operator*](#) (-=) to subtract any value. There are also operators for multiplication and assignment (*=) and division and assignment (/=). You will see how mathematical operations work in more detail in [Chapter 5](#).

Kotlin’s Built-In Types

You have seen variables that are of the **Int** type, and you have used the **String** type when calling `println`. Kotlin also has types to handle values like true/false, lists of elements, and key-value pairs for defining mappings of elements. [Table 2.1](#) shows many of the commonly used built-in types available in Kotlin:

Table 2.1 Commonly used built-in types

Type	Description	Examples
String	Text	"Madrigal" "happy meal"
Char	Single character	'x' Unicode character U+0041
Boolean	True/false values	true false
Int	Whole numbers	5 "Madrigal".length
Double	Decimal numbers	3.14 2.718
List	Collections of elements	3, 1, 2, 4, 3 "root beer", "club soda", "coke"
Set	Collections of unique elements	"Larry", "Moe", "Curly" "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"
Map	Collections of key-value pairs	"small" to 5.99, "medium" to 7.99, "large" to 10.99

If you have not seen all these types, do not be concerned – you will learn about them throughout the course of this book. In particular, you will learn more about strings in [Chapter 6](#) and numbers in [Chapter 5](#), and you will learn about lists, sets, and maps, together called [collection types](#), in [Chapter 9](#) and [Chapter 10](#).

Read-Only Variables

So far, you have seen variables whose values can be reassigned – also known as *mutable* variables. But often, you will want to use variables whose values should not change. For example, in the text adventure game, the player’s name will not change after it has been initially assigned.

Kotlin provides a different syntax for declaring [read-only](#) variables – variables that cannot be modified once they are assigned.

You declare a variable that can be modified using the `var` keyword. To declare a read-only variable, you use the `val` keyword.

Colloquially, variables whose values can change are referred to as `vars` and read-only variables are referred to as `vals`. We will follow this convention from now on, because “variable” and “read-only variable” are less distinct. `vars` and `vals` are both considered “variables,” so we will continue to use that term to refer to them as a group.

Add a `val` definition for the player’s name and print it after the initial player level:

Listing 2.6 Adding a `heroName` `val` (`Main.kt`)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

Run the program by clicking the run button, either next to the **main** function or in the menu bar. You will see the values of `playerLevel` and `heroName` printed in the console:

```
The hero announces her presence to the world.
Madrigal
4
The hero embarks on her journey to locate the enchanted sword.
5
```

Next, try reassigning `heroName` to a different **String** value, using the `=` assignment operator, and run the program again.

Listing 2.7 Trying to change `heroName`’s value (`Main.kt`)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    heroName = "Estragon"

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

Run the program, and you will see the following compilation error:

```
e: Main.kt: (9, 5): Val cannot be reassigned
```

The compiler complains because you tried to modify a `val`. Once a `val` has been assigned, it can never be reassigned.

Delete the second assignment to fix the reassignment error:

Listing 2.8 Fixing the `val` reassignment error (`Main.kt`)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    heroName = "Estragon"

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

`vals` are useful for guarding against accidentally changing variables that should be read-only. For this reason, we recommend that you use a `val` any time you do not need a `var`.

IntelliJ can detect when a `var` can be made a `val` instead by analyzing your code statically. If a `var` is never changed, IntelliJ will suggest that you convert it to a `val`. We suggest you follow IntelliJ’s suggestion – unless you are about to write the code that reassigns the `var`. To see what IntelliJ’s suggestion looks like, change `heroName` to a `var`:

Listing 2.9 Changing heroName to be reassignable (Main.kt)

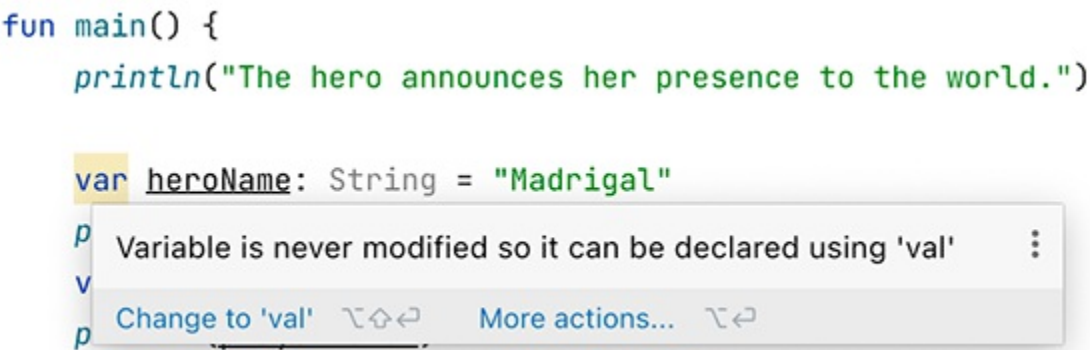
```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    var heroName: String = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

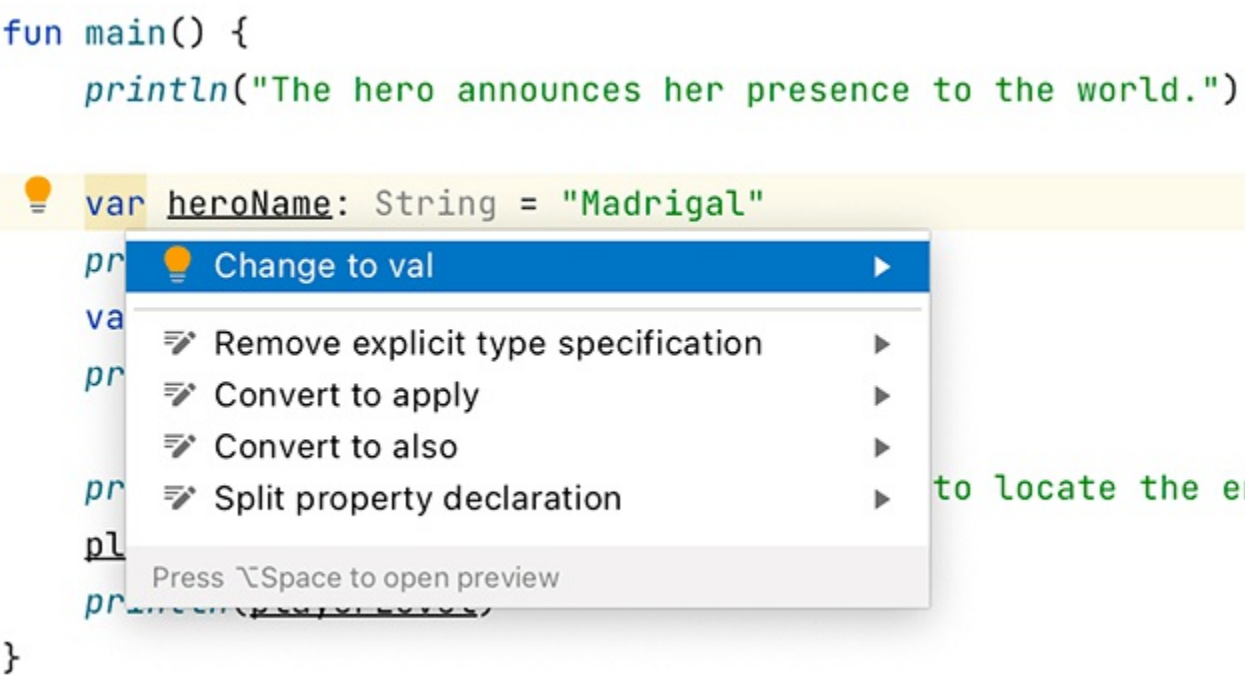
Because the value of heroName is never reassigned, it does not need to be (and should not be) a var. Notice that IntelliJ has added a mustard-colored highlight to the line with the var keyword. If you mouse over the var keyword, IntelliJ explains the suggested improvement (Figure 2.3).

Figure 2.3 Variable never modified tooltip



As expected, IntelliJ suggests converting heroName to a val. To apply the suggestion, click the var keyword next to heroName and press Option-Return (Alt-Enter). In the pop-up, select Change to val (Figure 2.4).

Figure 2.4 Making a variable immutable



IntelliJ automatically converts the var to a val:

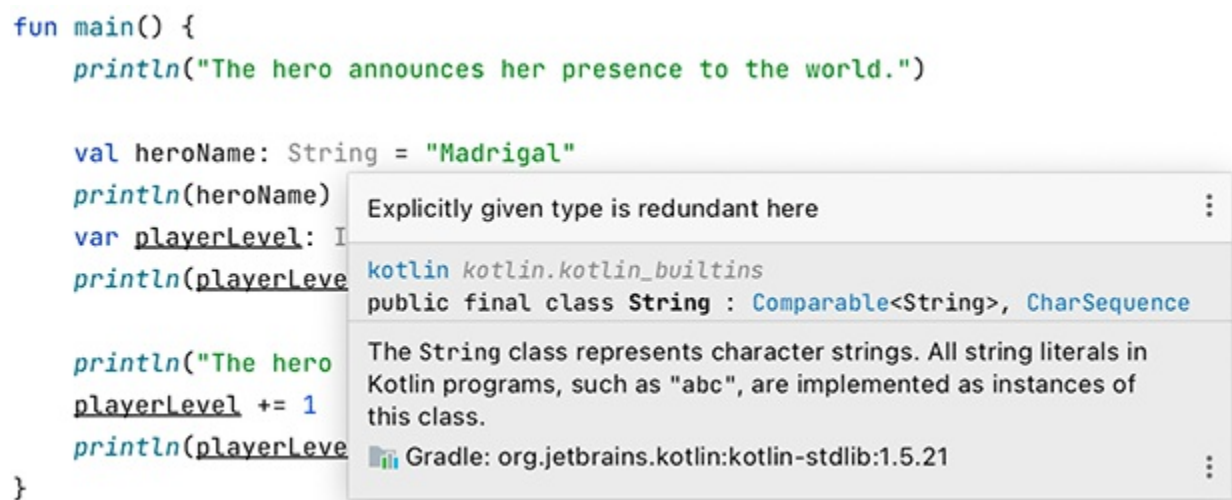
```
val heroName: String = "Madrigal"
println(heroName)
```

As we said earlier, we recommend that you use a val any time you can, so Kotlin can warn you about accidental reassignments. We also recommend that you pay attention to IntelliJ’s suggestions for code improvement. You may not always use them, but it is always worthwhile to take a look.

Type Inference

Notice that the type definitions you specified for the `heroName` and `playerLevel` variables are grayed out in IntelliJ. Grayed-out text indicates an element that is unnecessary or not being used. Mouse over the **String** type definition, and IntelliJ will provide an explanation for why the element is not required ([Figure 2.5](#)).

Figure 2.5 Redundant type information

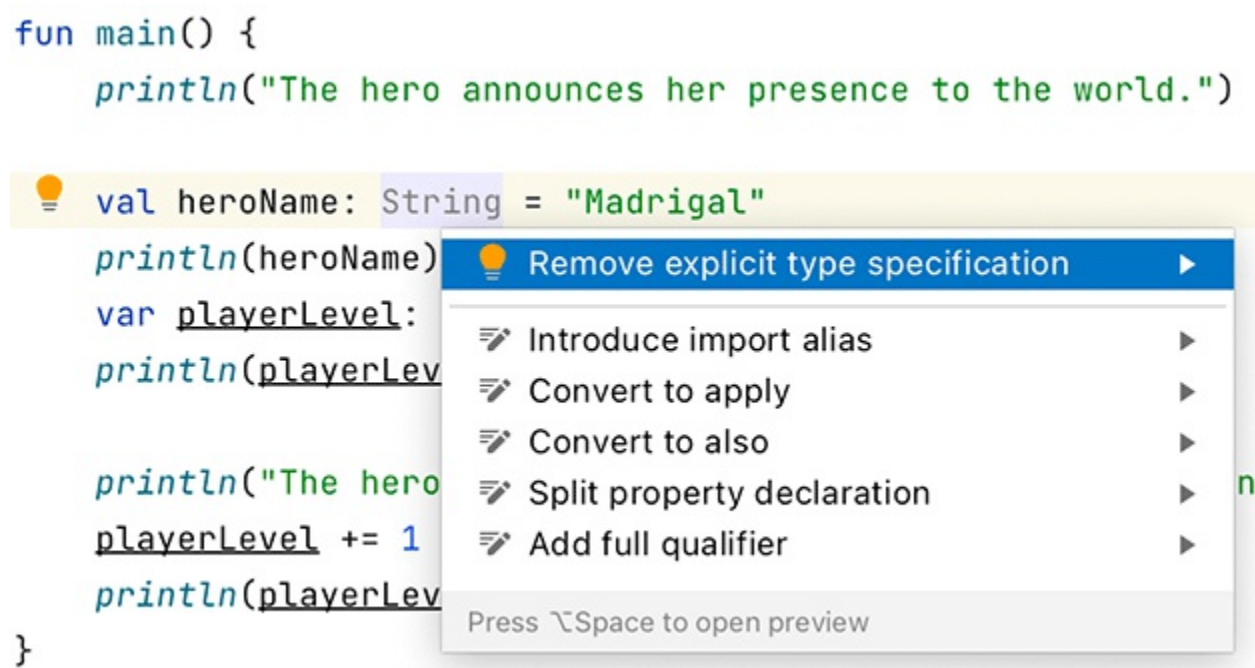


IntelliJ indicates that your type declaration is “redundant.” What does this mean?

Kotlin includes a feature called [type inference](#) that allows you to omit the type definition for variables that are assigned a value when they are declared. Because you assign data of the **String** type to `heroName` and of the **Int** type to `playerLevel` when you declare them, the Kotlin compiler infers the appropriate type information for both variables.

Just as IntelliJ can help you change a `var` to a `val`, it can also help you remove unneeded type specifications. Click the `String` type definition (`: String`) next to `heroName` and press Option-Return (Alt-Enter). Then click Remove explicit type specification in the pop-up ([Figure 2.6](#)).

Figure 2.6 Removing explicit type specification



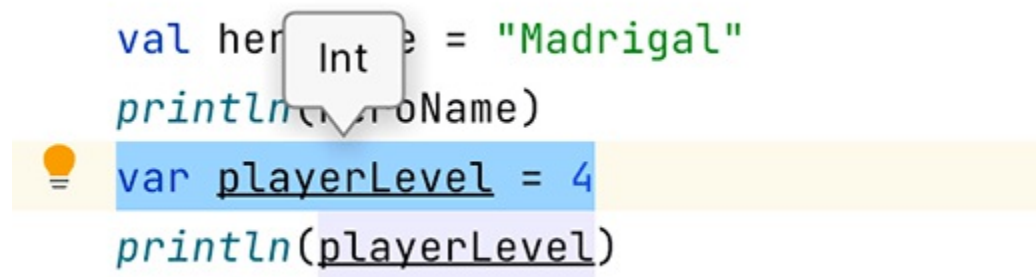
`: String` will disappear. Repeat the process for the `playerLevel` var to remove `: Int`.

Whether you take advantage of type inference or specify the type when you declare the variable, the compiler will keep track of the type. In this book, we use type inference where it is unambiguous to do so. Type inference helps keep code clean, concise, and easier to modify as your program changes.

By the way, IntelliJ will display the type of any variable on request, including those that use type inference. If you ever have a question about the type of a variable or expression, click its name or highlight a portion of your code and press View → Type Info (or use the keyboard shortcut Control-Shift-P [Ctrl-Shift-P]). IntelliJ will display its type ([Figure 2.7](#)).

Figure 2.7 Displaying the type info tooltip

```
val hero = "Madrigal"  
println(heroName)  
= var playerLevel = 4  
println(playerLevel)
```

A screenshot of a code editor showing four lines of code. The first line is 'val hero = "Madrigal"', the second is 'println(heroName)', the third is 'var playerLevel = 4', and the fourth is 'println(playerLevel)'. The third line is highlighted with a blue background. A tooltip box is positioned over the variable 'playerLevel', displaying the text 'Int'. To the left of the third line, there is a small orange lightbulb icon and an equals sign.

Compile-Time Constants

Earlier we told you that `vars` can have their values changed and `vals` cannot. That ... was a white lie. In fact, there are special cases where a `val` can return different values, which we will discuss in [Chapter 13](#). If you have a piece of data that you want to be absolutely, positively immutable – to never change – consider a [compile-time constant](#).

A compile-time constant must be defined outside any function, including **main**, because its value must be assigned at [compile time](#) (that is, when the program compiles) – hence the name. **main** and your other functions are called during [runtime](#) (when the program is executed), and the variables within them are assigned their values then. A compile-time constant exists before any of these assignments take place.

Compile-time constants also must be of one of the following basic types, because use of more complex types for a constant could jeopardize the compile-time guarantee. You will learn more about how types are constructed in [Chapter 14](#). Here are the supported basic types for a compile-time constant:

- **String**
- **Int**
- **Double**
- **Float**
- **Long**
- **Short**
- **Byte**
- **Char**
- **Boolean**

The name of your hero (Madrigal) will never change. Madrigal is the protagonist in this game, and players will not have the option to change her name. You can express this level of immutability in your code by extracting this value into a constant. In `Main.kt`, move the `heroName` variable above the declaration of the **main** function and add the `const` modifier:

Listing 2.10 Declaring a compile-time constant (`Main.kt`)

```
const val heroName = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    val heroName = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

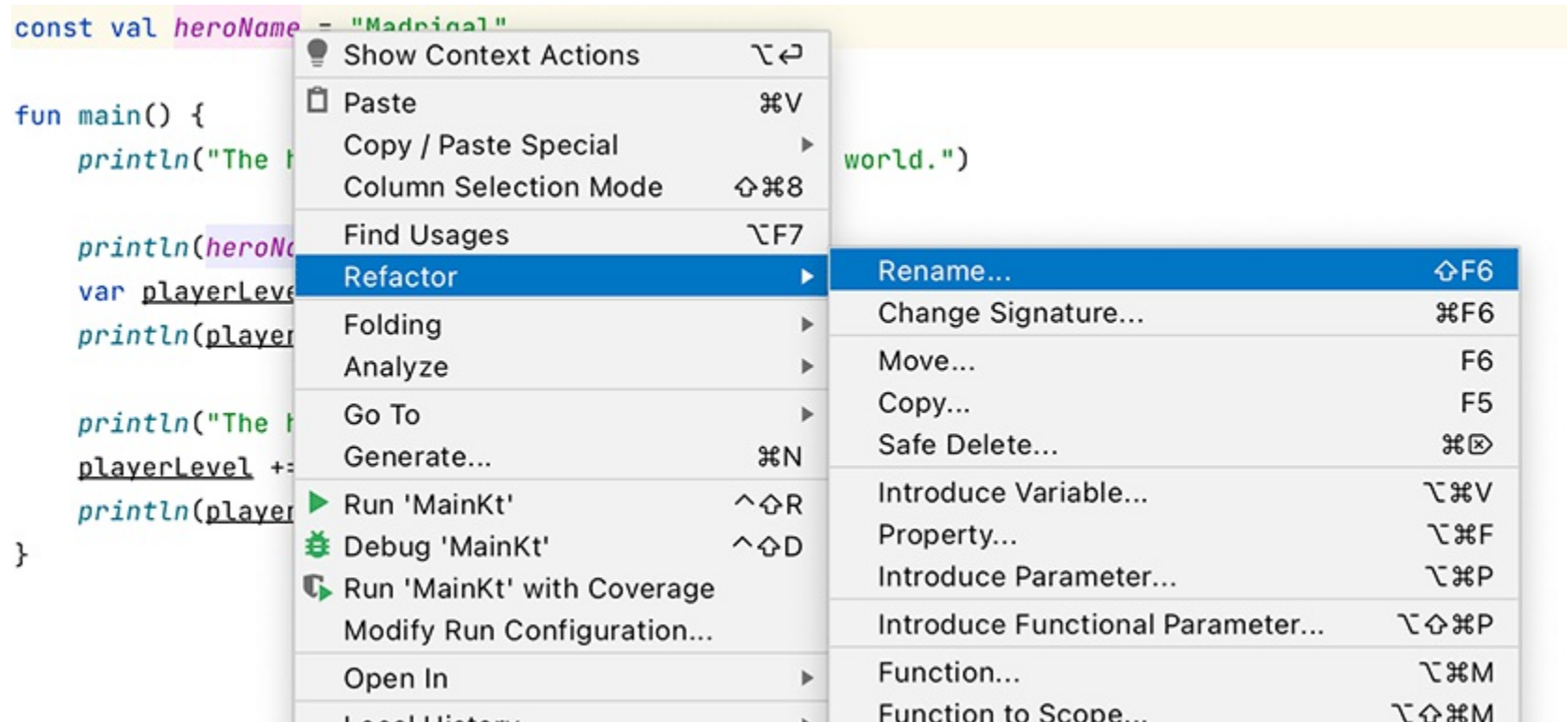
Prepending a `val` with the `const` modifier tells the compiler that it can be sure that this `val` will never change. In this case, the hero's name is guaranteed to have the string value "Madrigal", no matter what. This gives the compiler the flexibility to perform optimization behind the scenes. Run your code again to confirm that the output is the same after making this change.

In Kotlin, the convention is to use camelCase for the names of variables and all-caps SNAKE_CASE for the names of constants. So, instead of calling your new constant `heroName`, it would be more typical to call it `HERO_NAME`.

Since your project is small, you could easily change the name and update the one reference in your **main** function. However, on larger projects you will find this task to be quite a burden. Luckily, IntelliJ includes refactoring tools to make project-wide code changes on your behalf.

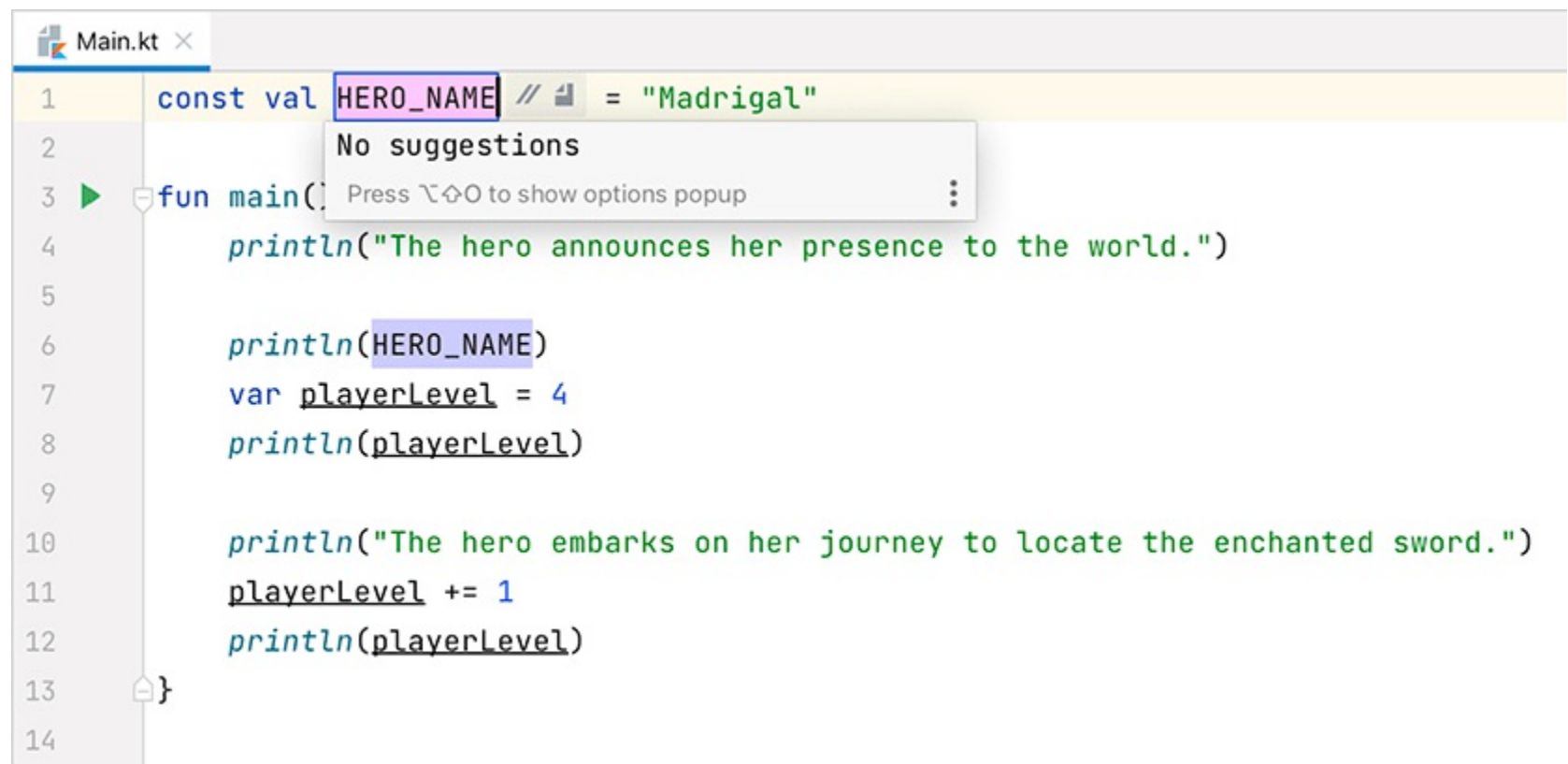
Right-click the `heroName` constant, then choose Refactor → Rename... ([Figure 2.8](#)).

Figure 2.8 The Refactor → Rename... menu item



The constant's name will be highlighted. Type `HERO_NAME`, replacing the highlighted text. As you type, IntelliJ will continue to highlight the constant, as shown in [Figure 2.9](#).

Figure 2.9 Renaming a constant



As you type, notice that the line `println(heroName)` updates with the new name. IntelliJ will find all usages of the constant in your project and update them with the new name. This will only affect one line of code in `bounty-board`, but in a large project IntelliJ could automatically update many files.

When you have finished entering the new name, press Return to confirm the change.

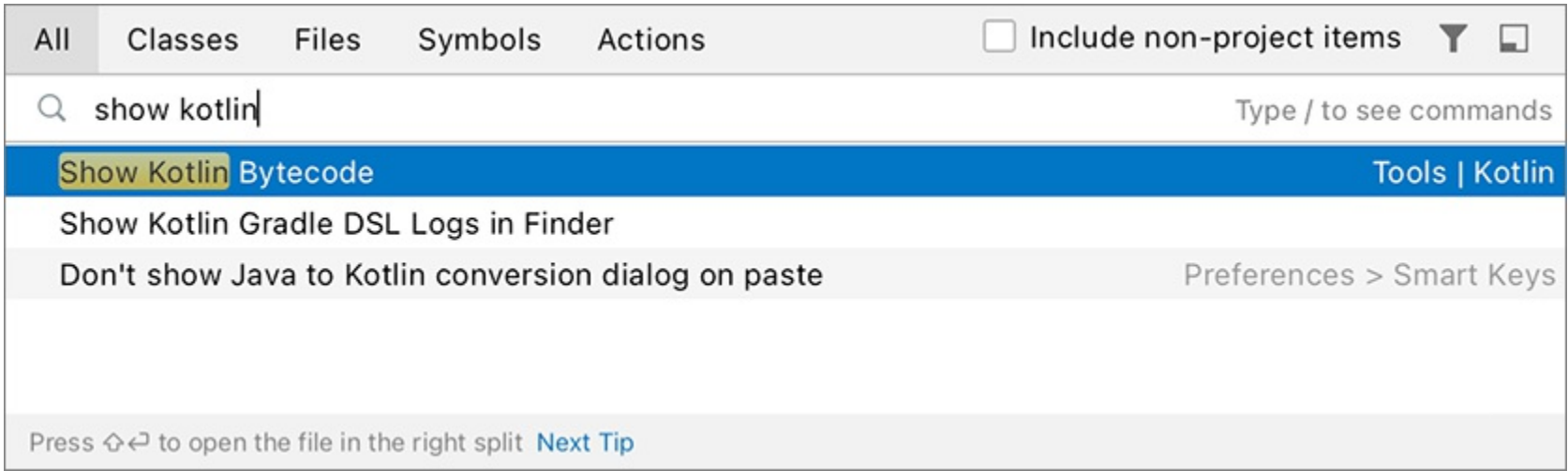
Inspecting Kotlin Bytecode

You learned in [Chapter 1](#) that Kotlin can be used to write programs that run on the JVM, where Java bytecode is executed. When targeting the JVM, it is often useful to inspect the Java bytecode that the Kotlin compiler generates to run on the JVM. You will look at the bytecode in several places in this book as a way to analyze how a particular language feature works on the JVM.

Knowing how to inspect the Java equivalent of the Kotlin code you write is a great technique for understanding how Kotlin works, especially if you have Java experience. If you do not have Java experience specifically, the Java code will likely share familiar traits with a language that you have worked with – so think of it as a pseudocode to aid your understanding. And if you are brand new to programming – congratulations! In choosing Kotlin, you have chosen a language that, as you are about to see, allows you to express the same logic that Java does, typically in much less code.

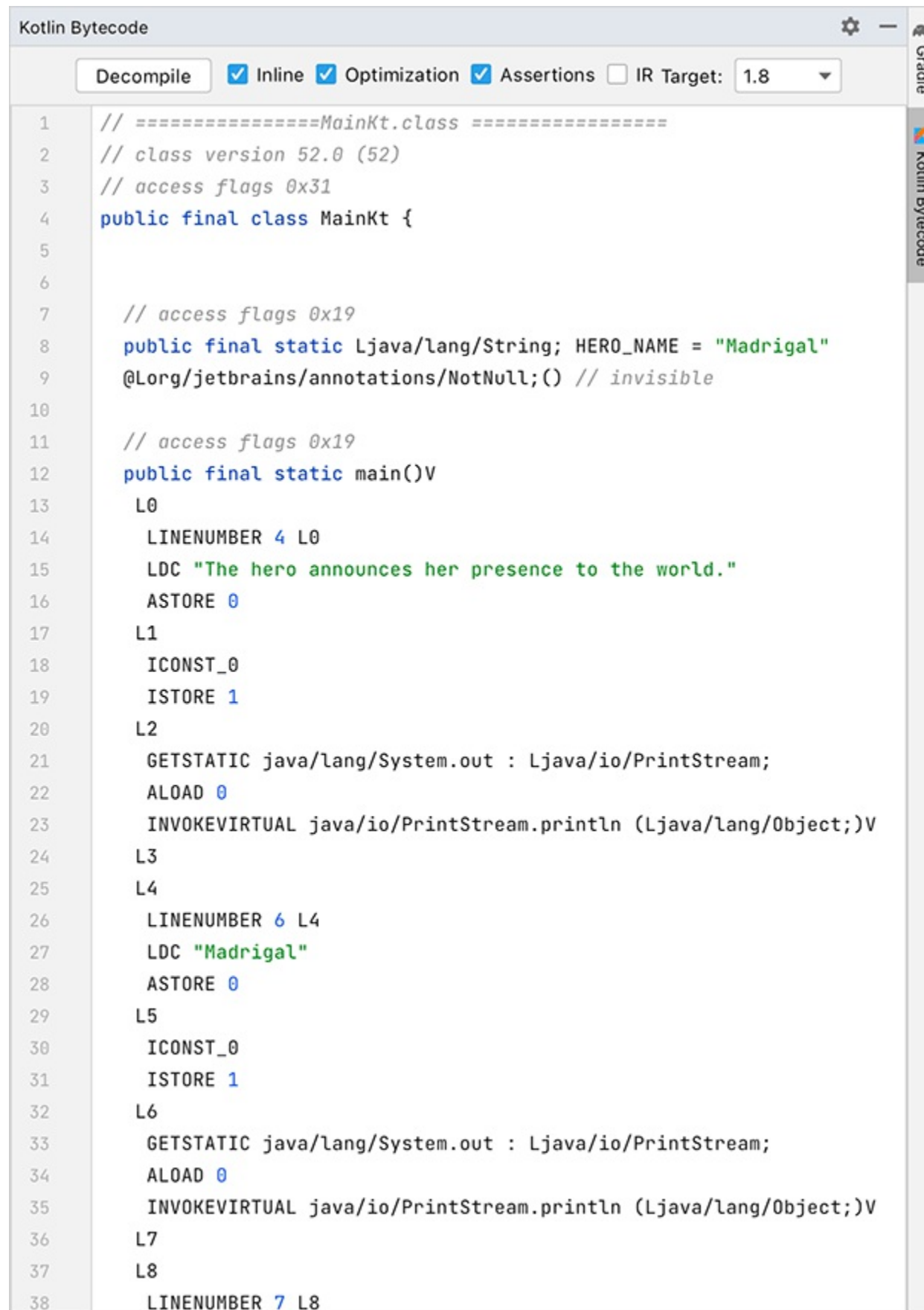
For example, you may have wondered how using type inference when defining variables in Kotlin affects the resulting bytecode that is generated to run on the JVM. To find out, you can use the Kotlin bytecode tool window. In `Main.kt`, press the Shift key twice to open the Search Everywhere dialog. Begin entering “show kotlin bytecode” in the search box and select Show Kotlin Bytecode from the list of available actions when it appears ([Figure 2.10](#)).

Figure 2.10 Showing Kotlin bytecode



The Kotlin bytecode tool window will open ([Figure 2.11](#)). (You can also open the tool window with Tools → Kotlin → Show Kotlin Bytecode.)

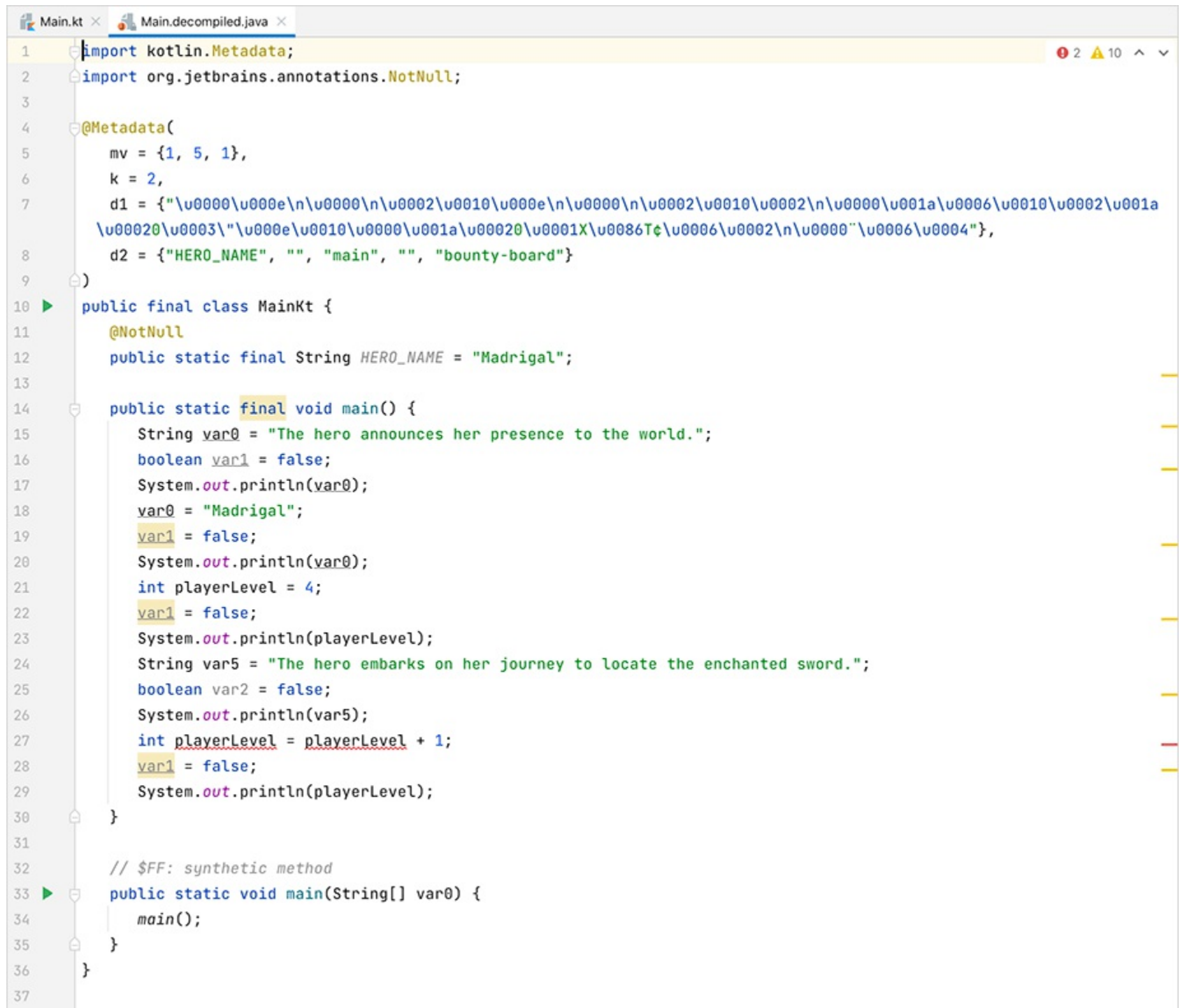
Figure 2.11 Kotlin bytecode tool window



If bytecode is not your native tongue, fear not. You can translate the bytecode into Java to see it represented in terms you may be more familiar with. In the bytecode tool window, click the **Decompile** button at the top left.

A new tab will open showing `Main.decompiled.java` ([Figure 2.12](#)), a Java version of the bytecode the Kotlin compiler generated for the JVM.

Figure 2.12 Decompiled bytecode



(Occasionally, the bytecode inspector will show red underlines in the decompiled code. This is because of a quirk in the interaction between Kotlin and Java, rather than a problem – you can safely ignore warnings and errors that appear in generated Java bytecode.)

Find the variable declaration for `playerLevel`:

```
int playerLevel = 4;
```

Although you omitted type declarations from the definitions of both variables in the Kotlin source, the bytecode that was generated includes explicit type definitions. This is how the variables would be declared in Java, and the bytecode gives a behind-the-scenes look at Kotlin's type inference support.

You will dig deeper into the decompiled Java bytecode in later chapters. For now, close

`Main.decompiled.java` (using the `x` in its tab) and the bytecode tool window (using the  icon at the top right).

Unfortunately, this tool only works when your Kotlin code targets the JVM. Kotlin/JS code transpiles into JavaScript, and Kotlin/Native code compiles down into native machine code. For both of the other targets, there is no bytecode to inspect. We encourage you to take advantage of this tool throughout the book while you are targeting the JVM. It can be invaluable while learning Kotlin – though you may find yourself using it less and less as you get more comfortable with the language.

In this chapter, you have learned how to store basic data in `vars` and `vals` and seen when to use each, depending on whether you need to be able to change their values. You have seen how to declare immutable values using compile-time constants. Last, you learned how Kotlin leverages the power of type inference to save you keystrokes every time you declare a variable. You will be using all these basic tools over and over as you proceed through this book.

In the next chapter, you will learn how to represent more complex states using conditionals.

For the More Curious: Java Primitive Types in Kotlin

In Java, there are two kinds of types: reference types and primitive types. Reference types have a matching source code definition. Some reference types are also called “boxed” or “object” types. Java also offers primitive types (often called just “primitives”), which have no source file definition and are represented by special keywords instead.

The names of reference types in Java begin with a capital letter to indicate that they are backed by a source definition. Here is `playerLevel` defined using a Java reference type:

```
Integer playerLevel = 4;
```

The names of Java primitive types start with a lowercase letter:

```
int playerLevel = 4;
```

All primitives in Java have a corresponding reference type. (But not all reference types have a corresponding primitive type.) Why use one versus the other?

One reason for choosing a reference type is that there are certain features of the Java language that are only available when using reference types. Generics, for example, which you will learn about in [Chapter 18](#), do not work with primitives. Reference types can also work with the object-oriented features of Java more readily than Java primitives. (You will learn about object-oriented programming and the object-oriented features of Kotlin in [Chapter 13](#).)

On the other hand, primitives offer better performance and some other perks.

Unlike Java, Kotlin provides only one kind of type: reference types.

```
var playerLevel: Int = 4
```

Kotlin made this design decision for several reasons. First, if there is no choice between kinds of types, you cannot code yourself into a corner as easily as you can with multiple kinds to choose from. For example, what if you define an instance of a primitive type, then realize later that you need to use the generic feature, which requires a reference type? Having only reference types in Kotlin means you will never encounter this problem.

If you are familiar with Java, you may be thinking, “But primitives offer better performance than reference types.” This is true. But take another look at the `playerLevel` variable in the decompiled bytecode you saw earlier:

```
int playerLevel = 4;
```

As the lowercase `int` indicates, a primitive type was used in place of the reference type. Why is that, if Kotlin only has reference types? The Kotlin compiler will, where possible, use primitives in the Java bytecode, because they do indeed offer better performance.

Kotlin gives you the ease of reference types with the performance of primitives under the hood. In Kotlin, you will find a corresponding reference type for the eight primitive types you may be familiar with in Java.

Challenge: hasSteed

In [Chapter 1](#), your challenge was to try out some mathematical operations in the Kotlin REPL. Most of the challenges in the rest of the book are based on the project you have been working on – in this case, bounty-board. Before beginning a challenge, make a copy of your project and attack the challenge in that copy. Many chapters build on previous chapters, and working on challenges in a copy of the project ensures that you will be able to progress through the book.

Here is your first challenge based on bounty-board: In the text adventure game, players can acquire a dragon or minotaur to ride. Define a variable called `hasSteed` to track whether the player has acquired one. Give the variable an initial state indicating that the player does not have one yet.

Remember to create a copy of bounty-board before making these changes. The next chapter will continue to build out bounty-board and is not expecting your players to have steeds.

Challenge: The Unicorn’s Horn

Imagine this scene from the adventure game:

The hero Madrigal arrives at a pub known as The Unicorn’s Horn. The publican asks, “Do you need to stable a steed?”

“No,” Madrigal replies, “I have no steed. But I do have 50 gold, and I would like a drink.”

“Excellent,” says the publican. “I have mead, wine, and LaCroix. What will you have?”

For this challenge, add below your `hasSteed` variable the variables required for the scene at The Unicorn’s Horn, using type inference and assigned values where possible. Add variables to hold values for the name of the pub, the name of the current publican on duty, and how much gold the player has acquired so far.

Notice that The Unicorn’s Horn has a menu of drinks the hero can select from. What type might you use to represent the menu? If you need to, consult [Table 2.1](#).

Challenge: Magic Mirror

Refreshed, Madrigal is ready for a challenging quest. Are you?

The hero discovers a magic mirror that shows a player the reflection of their `HERO_NAME`. Using the **`String`** type's magic, transform the `HERO_NAME` string "Madrigal" into "lagirdaM", a reflection of its value.

To solve this challenge, consult the documentation for **`String`** at kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/index.html. You will find that, fortunately, the actions that a particular type can perform are usually very intuitively named (hint).

Part II

Basic Syntax

Variables and print statements will only get you so far in a program. The next five chapters will introduce fundamental components that appear in virtually every Kotlin application.

You will expand bounty-board, using all the language features that you would expect to see in a basic application. If you are not new to programming, there is a good chance that you have seen these language features in another programming language. Even so, there is still much to learn about how these concepts behave in the world of Kotlin.

Conditionals

In this chapter, you will learn how to define rules for when code should be executed. This language feature is called [control flow](#), and it allows you to describe the conditions for when specific portions of your program should run. You will see the `if/else` statement and expression and the `when` expression, and you will learn how to write true/false tests using the comparison and logical operators.

To see these features in action, you will continue working in the bounty-board project, creating the quests that will be presented to the player. The quests will scale in difficulty as the player gets stronger, which is where conditionals will come into play.

if/else Statements

For the purposes of your bounty board, the player’s strength is determined by their `playerLevel`, which you created in the previous chapter. Lower values indicate that the hero is closer to the beginning of their epic journey, and higher values indicate that a hero has gained experience and has become stronger.

Your goal is to present quests based on the player’s current level. For example, if a player is at level 1 (the starting point for new characters), then they need to be given an easy quest, like a tutorial.

Within the **main** function, write your first if/else statement, as shown below. We will break down this new code after you enter it.

Listing 3.1 Printing the player’s quest (Main.kt)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else {
        println("Locate the enchanted sword.")
    }

    println("The hero embarks on her journey to locate the enchanted sword.")
    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

Let’s go through this new code, top to bottom.

First, you add an if/else statement. After the `if` keyword, you specify a condition in parentheses. The condition here poses the following true/false question: “Does the player have a `playerLevel` of 1?” You express this with the `==` [structural equality operator](#). It can be read as “is equal to,” so this statement reads “If `playerLevel` is equal to 1.”

Your `if` statement is followed by a statement in curly braces (`{}`). The code within the curly braces is what you want the program to do if the `if` condition evaluates as the Boolean value `true` – in this case, if `playerLevel` has a value of exactly 1.

```
if (playerLevel == 1) {
    println("Meet Mr. Bubbles in the land of soft things.")
}
```

Included in this statement is the familiar **println** function used to print something to the console. In short, your if/else statement so far says that if Madrigal is at level 1, the program should print a beginner-level quest.

(While your `if` statement’s curly braces enclose only one statement, more can be included if you want multiple actions to be taken when the `if` evaluates as `true`.)

What if `playerLevel` has a value other than 1? In that case, the `if` evaluates as `false`, and your program will skip the expressions in curly braces that follow the `if` and move on to the `else`. Think of `else` as meaning “otherwise”: If some condition is true, do this; *otherwise* do that. `else` – like `if` – is followed by a set of expressions in curly braces that tell the compiler what to do. But – unlike `if` – it does not need to define a condition. It applies whenever the `if` does not, so the curly braces immediately follow the keyword.

```
else {
    println("Locate the enchanted sword.")
}
```

The `else` block is optional, from the perspective of the Kotlin language. You are allowed to have an `if` statement without an `else` branch, which you will see in later chapters. In that case, when the `if` evaluates as `false`, your program will execute whatever follows the `if` statement. You can also declare an `else` block with an empty body, which works the same way.

The only difference between the branches in your code is that the call to **println** contains a different quest. Instead of giving the player a quest that would be trivial for their progress in the game, the `else` branch asks the player to “Locate the enchanted sword.” (Thus far, most of the function calls that you have seen serve only to print strings out to the console. You will learn more about functions, including how to define your own, in [Chapter 4](#).)

Putting this all together in plain English, your code says to the compiler, “If the hero is at level 1, print `Meet Mr. Bubbles in the land of soft things.` to the console. If the hero is not at level 1, print `Locate the enchanted sword.` to the console.”

The structural equality operator, `==`, is a [comparison operator](#). [Table 3.1](#) lists Kotlin’s comparison operators. You do not need to know all the operators listed now, as you will learn more about them later. Return to this table when

you are considering possible operators to express a condition.

Table 3.1 Comparison operators

Operator	Description
<	Evaluates whether the value on the left is less than the value on the right.
<=	Evaluates whether the value on the left is less than or equal to the value on the right.
>	Evaluates whether the value on the left is greater than the value on the right.
>=	Evaluates whether the value on the left is greater than or equal to the value on the right.
==	Evaluates whether the value on the left is equal to the value on the right.
!=	Evaluates whether the value on the left is not equal to the value on the right.
===	Evaluates whether the two instances point to the same reference.
!==	Evaluates whether the two instances do not point to the same reference.

Back to business. Run `Main.kt` by clicking the run button to the left of the **main** function. You should see this output:

```
The hero announces her presence to the world.
Madrigal
4
Locate the enchanted sword.
Time passes...
The hero returns from her quest.
5
```

Since the condition you defined, `playerLevel == 1`, is false, the `if` branch in the `if/else` statement was skipped, and the `else` branch executed instead. (We use the word *branch* because the flow of execution branches depending on whether the specified condition is met.) Now, try changing the `playerLevel` value to 1:

Listing 3.2 Modifying `playerLevel` (`Main.kt`)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
var playerLevel = 4
    var playerLevel = 1
    println(playerLevel)

    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else {
        println("Locate the enchanted sword.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

Run the program again, and you will see:

```
The hero announces her presence to the world.
Madrigal
1
Meet Mr. Bubbles in the land of soft things.
Time passes...
The hero returns from her quest.
2
```

Now, the condition you defined is true (`playerLevel` is equal to 1), so the `if` branch is triggered.

Adding more conditions

The quest determination code gives a crude idea of what activities the hero should take on, but it is ... well, crude. The tutorial level is a good starting point, but for any player level greater than 1, there is only one quest – locating the enchanted sword. And once you have one enchanted sword, you do not need another.

To make your `if/else` statement more nuanced, you can add more conditions to check for and more branches to include as possible results. You do this with `else if` branches, whose syntax is just like an `if` branch’s, but placed between the `if` and the `else`. Update your `if/else` statement to include four `else if` branches checking for intermediate values of `playerLevel`. While you are at it, change the value assigned to `playerLevel` back to 4:

Listing 3.3 Checking for more player conditions (`Main.kt`)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
var playerLevel = 1
    var playerLevel = 4
    println(playerLevel)

    if (playerLevel == 1) {
```

```
println("Meet Mr. Bubbles in the land of soft things.")
} else if (playerLevel <= 5) {
    println("Save the town from the barbarian invasions.")
} else if (playerLevel == 6) {
    println("Locate the enchanted sword.")
} else if (playerLevel == 7) {
    println("Recover the long-lost artifact of creation.")
} else if (playerLevel == 8) {
    println("Defeat Nogartse, bringer of death and eater of worlds.")
} else {
    println("Locate the enchanted sword.")
    println("There are no quests right now.")
}

println("Time passes...")
println("The hero returns from her quest.")

playerLevel += 1
println(playerLevel)
}
```

Your new logic reads like this:

If Madrigal is at this level ...	... print this message
1	Meet Mr. Bubbles in the land of soft things.
2-5	Save the town from the barbarian invasions.
6	Locate the enchanted sword.
7	Recover the long-lost artifact of creation.
8	Defeat Nogartse, bringer of death and eater of worlds.
9+	There are no quests right now.

Run the program again. Because the value of Madrigal’s `playerLevel` is 4, the first `if` will evaluate as false, and its branch will not be executed. But `else if (playerLevel <= 5)` is true, so you will see `Save the town from the barbarian invasions.` in the console.

The compiler evaluates the conditions of an `if/else` from top to bottom and stops checking conditions as soon as one evaluates as true. If none of the conditions you provide are true, the `else` branch will be executed.

This means that the order of the conditions matters: If you had checked `playerLevel <= 5` before checking `playerLevel == 1`, then the level 1 quest would never be shown. (Do not make this change to your code. It is only for illustration.)

```
if (playerLevel <= 5) {           // Triggered for any value 5 or less
    println("Save the town from the barbarian invasions.")
} else if (playerLevel == 1) {    // Only triggered for a value of 1
    println("Meet Mr. Bubbles in the land of soft things.")
} else if (playerLevel == 6) {
    println("Locate the enchanted sword.")
} else if (playerLevel == 7) {
    println("Recover the long-lost artifact of creation.")
} else if (playerLevel == 8) {
    println("Defeat Nogartse, bringer of death and eater of worlds.")
} else {
    println("There are no quests right now.")
}
```

In this example, any `playerLevel` less than or equal to 5 would trigger the first condition, but only a value of 1 would trigger the second branch. Because the first `if` condition would match, the `else if (playerLevel == 1)` branch would never be considered.

You have added more subtlety in how the player’s level is reported by including `else if` statements with more conditions to check when the initial `if` condition evaluates as false. Try varying `playerLevel`’s value to trigger the result in each branch you defined. When you are done, return `playerLevel` to a value of 4.

Nested if/else statements

One of the quests on the bounty board, “Save the town from the barbarian invasions,” is fairly abstract. There are several ways to achieve the desired outcome, with diplomacy being one of them. If the player is on good terms with the leader of a barbarian tribe, they may be able to engage in a friendly discussion with the tribe to clear up the misunderstanding.

To make this possibility more apparent to the player, the quest’s name will change depending on whether the player is friends with the barbarians. Before you can update your quest determination logic, you will need to add a variable to track whether this barbarian friendship exists. (What type do you think it will be?)

After defining the variable to track your barbarian friendship, you will again tweak your `if/else` statement. When the player’s level is 2 to 5, you can use an additional nested `if/else` to print the correct quest title. (As you enter the changes below, do not miss the added `}` before `else if (playerLevel == 6).`)

Listing 3.4 Checking for barbarian friendships (Main.kt)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
}
```

```
println(playerLevel)

val hasBefriendedBarbarians = true
if (playerLevel == 1) {
    println("Meet Mr. Bubbles in the land of soft things.")
} else if (playerLevel <= 5) {
    if (hasBefriendedBarbarians) {
        println("Convince the barbarians to call off their invasion.")
    } else {
        println("Save the town from the barbarian invasions.")
    }
} else if (playerLevel == 6) {
    println("Locate the enchanted sword.")
} else if (playerLevel == 7) {
    println("Recover the long-lost artifact of creation.")
} else if (playerLevel == 8) {
    println("Defeat Nogartse, bringer of death and eater of worlds.")
} else {
    println("There are no quests right now.")
}

println("Time passes...")
println("The hero returns from her quest.")

playerLevel += 1
println(playerLevel)
}
```

You added a Boolean `val` representing whether the player has befriended the barbarians and an `if/else` statement to create a new output when barbarians are befriended *and* the player’s level is between 2 and 5. Recall that `playerLevel` has a value of 4, so you should expect to see the new message when you run the program. Run it and see. Your output should be:

```
The hero announces her presence to the world.
Madrigal
4
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

If you see any other output, check that your code matches [Listing 3.4](#) exactly – in particular that `playerLevel` is assigned a value of 4.

Nesting conditionals allows you to create logical branches within branches so the conditions that you check can be precise and complex.

More elegant conditionals

If you do not keep a sharp eye on them, conditionals will explode all over the place like tribbles. Thankfully, Kotlin allows you to take advantage of conditionals’ usefulness while keeping them concise and readable. Let’s look at some examples.

Logical operators

In the bounty board, more complex conditions can arise that you need to check. For example, the barbarian diplomatic approach may be available if the player has befriended the barbarians *or* if they are themselves a barbarian – or it may be blocked if they have angered the tribe.

You could use a series of `if/else` statements to determine which quest to show, but you would end up with a lot of duplicate code, and the logic of the conditions would be masked. There is a more elegant and reader-friendly way: using logical operators in a conditional.

Add two new variables and update your nested `if` statement’s condition to enhance the quest logic:

Listing 3.5 Using logical operators in a conditional (`Main.kt`)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        if (hasBefriendedBarbarians) {
        // Check whether diplomacy is an option
        if (!hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")) {
            println("Convince the barbarians to call off their invasion.")
        } else {
            println("Save the town from the barbarian invasions.")
        }
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
}
```

```
println(playerLevel)
}
```

You added two `vals` called `hasAngeredBarbarians` and `playerClass` to track the new components of this condition. (They are read-only because you will not change them at runtime in `bounty-board`.) That part is familiar, but there are a couple of new things going on, too. First, you included a [code comment](#), indicated by `//`.

Anything to the right of `//` is included in the comment and is ignored by the compiler, so you can use any syntax you want there. Comments are useful for organizing and adding information about your code, making it more readable for others (or for your future self, who may not remember all the details).

Next, you used several [logical operators](#) in your `if`. Logical operators allow you to combine comparison operators into a larger statement.

`!` is the [logical ‘not’ operator](#), which returns the opposite of a Boolean value: If the element it is appended to is true, the expression is false, and vice versa. `&&` is the [logical ‘and’ operator](#), and it requires that *both* the condition on its left *and* the condition on its right be true for the expression as a whole to be true. `||` is the [logical ‘or’ operator](#). It allows the expression as a whole to be true if *either* the condition on its left *or* the condition on its right (or both) is true.

[Table 3.2](#) shows Kotlin’s logical operators.

Table 3.2 Logical operators

Operator	Description
<code>&&</code>	Logical ‘and’: true if and only if both are true (false otherwise).
<code> </code>	Logical ‘or’: true if either is true (false only if both are false).
<code>!</code>	Logical ‘not’: returns the opposite of a Boolean value.

One note: When operators are combined, there is an order of precedence that determines the order they are evaluated in. Operators with the same precedence are applied from left to right. You can also group operations by surrounding the operators that should be evaluated as a group in parentheses. Here is the order of operator precedence, from highest to lowest:

- `!` (logical ‘not’)
- `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to)
- `==` (structural equality), `!=` (non-equality)
- `&&` (logical ‘and’)
- `||` (logical ‘or’)

Getting back to `bounty-board`, let’s take a look at your new condition:

```
if (!hasAngeredBarbarians &&
    (hasBefriendedBarbarians || playerClass == "barbarian")) {
    println("Convince the barbarians to call off their invasion.")
}
```

Put another way, if the player has *not* angered the barbarians *and* they either have become friends with the barbarians *or* they themselves are a barbarian, then the bounty board will present a diplomatic approach to preventing an invasion.

Madrigal has not angered the barbarians. She is not herself a barbarian, but she is friends with them. Thus, the condition is met, and the bounty board should tell Madrigal to talk to the barbarians. Run your program to check. You should see:

```
The hero announces her presence to the world.
Madrigal
4
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

Think about the nested conditional statements that would be required to express this logic without logical operators. These operators give you the tools to express complex logic clearly.

Logical operators are not only for conditionals. They can be used in many expressions, including in the declaration of a variable. Add a new Boolean variable that encapsulates the conditions required to converse with the barbarians and [refactor](#) (that is, rewrite without changing the behavior of) your conditional to use the new variable.

Listing 3.6 Using logical operators in the declaration of a variable (Main.kt)

```
...
fun main() {
    ...
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        // Check whether diplomacy is an option
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")

        if (!hasAngeredBarbarians &&
```



```

        (hasBefriendedBarbarians || playerClass == "barbarian")) {
    if (canTalkToBarbarians) {
        println("Convince the barbarians to call off their invasion.")
    } else {
        println("Save the town from the barbarian invasions.")
    }
} else if (playerLevel == 6) {
    println("Locate the enchanted sword.")
} else if (playerLevel == 7) {
    println("Recover the long-lost artifact of creation.")
} else if (playerLevel == 8) {
    println("Defeat Nogartse, bringer of death and eater of worlds.")
} else {
    println("There are no quests right now.")
}
...
}

```

You have moved the condition check to a new `val` called `canTalkToBarbarians` and changed your `if/else` statement to check its value. This is functionally equivalent to what you had written before, but now you express the rules as a value assignment instead. The name of the value clearly signifies what the rule you defined expresses in “human-readable” terms: whether or not the player is on speaking terms with the barbarians. This is an especially useful technique for when your program’s rules become complex, and it helps to communicate what your rules mean for future readers of your code.

Run your program again to make sure it functions as before. The output should be the same.

Conditional expressions

Now the `if/else` statement displays an appropriate quest correctly – and with some subtlety.

On the other hand, it would be somewhat unwieldy to make changes to it, because each branch repeats a similar `println` statement. What if you wanted to make a change to the overall formatting of how the quests are displayed? The program in its current state would require you to hunt through each branch in the `if/else` statement and change each `println` function to the new format.

You can solve this by changing the `if/else` statement you wrote to a [conditional expression](#) instead. A conditional expression is like a conditional statement, except that you assign the `if/else` to a value that you can use later. Use a conditional expression in your barbarian quest branch to see what this looks like.

Listing 3.7 Using a conditional expression (Main.kt)

```

...
fun main() {
    ...
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        // Check whether diplomacy is an option
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")

        val barbarianQuest: String = if (canTalkToBarbarians) {
            println("Convince the barbarians to call off their invasion.")
            "Convince the barbarians to call off their invasion."
        } else {
            println("Save the town from the barbarian invasions.")
            "Save the town from the barbarian invasions."
        }
        println(barbarianQuest)
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }
    ...
}

```

Through the `if/else` expression, the new variable `barbarianQuest` is assigned a string value from one of the cases in your `if` statement depending on what the value of `canTalkToBarbarians` is. That is the beauty of a conditional expression. Because you can now print the barbarian quest using the `barbarianQuest` variable, you can use a single `println` call to handle both cases.

You can clean up your quest logic even further by making the same change to your complex `if/else` statement. Refactor your quest determination logic and watch as six virtually identical print statements disappear.

Listing 3.8 Determining quests with a conditional expression (Main.kt)

```

...
fun main() {
    ...
    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    val quest: String = if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
        "Meet Mr. Bubbles in the land of soft things."
    } else if (playerLevel <= 5) {
        // Check whether diplomacy is an option
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")

        val barbarianQuest: String = if (canTalkToBarbarians) {
            "Convince the barbarians to call off their invasion."
        } else {
            "Save the town from the barbarian invasions."
        }
        println(barbarianQuest)
    } else if (playerLevel == 6) {

```

```

        println("Locate the enchanted sword.")
        "Locate the enchanted sword."
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
        "Recover the long-lost artifact of creation."
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
        "Defeat Nogartse, bringer of death and eater of worlds."
    } else {
        println("There are no quests right now.")
        "There are no quests right now."
    }
}

println("The hero approaches the bounty board. It reads:")
println(quest)

println("Time passes...")
println("The hero returns from her quest.")

playerLevel += 1
println(playerLevel)
}

```

(Incidentally, if you are tired of keeping your code nicely indented as you make changes, IntelliJ is here to help. Select Code → Auto-Indent Lines and enjoy the simple pleasure of clean indents.)

When you need to assign a variable based on a condition, you can likely use a conditional expression. Keep in mind, however, that conditional expressions are often most intuitive when the value being assigned from each branch is of the same type (like the `quest` strings).

Run your code one more time to make sure everything works as expected. You should see some familiar output (with one addition), but your code is now more elegant and reader-friendly.

```

The hero announces her presence to the world.
Madrigal
4
The hero approaches the bounty board. It reads:
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5

```

Removing braces from if/else expressions

In cases where you have a single response for the matching condition, it is valid (at least, syntactically – more on that shortly) to omit the curly braces wrapping the expression. You can only omit the `{}`s when a branch contains exactly one expression. Omitting them from a branch with more than one expression will affect how the code is evaluated, and Kotlin disallows an `if` statement without a statement or a pair of curly braces.

Take a look at a version of `quest` without braces:

```

val quest: String = if (playerLevel == 1)
    "Meet Mr. Bubbles in the land of soft things."
else if (playerLevel <= 5) {
    // Check whether diplomacy is an option
    val canTalkToBarbarians = !hasAngeredBarbarians &&
        (hasBefriendedBarbarians || playerClass == "barbarian")

    if (canTalkToBarbarians) "Convince the barbarians to call off their invasion."
    else "Save the town from the barbarian invasions."
} else if (playerLevel == 6) "Locate the enchanted sword."
else if (playerLevel == 7) "Recover the long-lost artifact of creation."
else if (playerLevel == 8)
    "Defeat Nogartse, bringer of death and eater of worlds."
else "There are no quests right now."

```

This version of the `quest` conditional expression does the same thing as the version you have in your code. It even expresses the same logic in fewer lines of code. But which version do you find easier to read and understand at a glance? If you chose the version with the braces – the version in your code – you have chosen the style that the Kotlin community prefers.

We recommend that you do not omit braces for conditional statements or expressions that span more than one line. For one thing, without braces it becomes increasingly difficult to understand where a branch starts and ends with every condition that is added. For another, omitting the braces for the conditional increases the risk of a new contributor mistakenly updating the wrong branch or misunderstanding what the implementation does. It is just not worth it to save a few keystrokes.

Also, while the code above expresses the same thing with or without braces, this is not the case for every example. If you have multiple expressions on a branch and you drop the braces around the conditional, only the first expression is executed in that branch. Take this example:

```

var arrowsInQuiver = 2
if (arrowsInQuiver >= 5) {
    println("Plenty of arrows")
    println("Cannot hold any more arrows")
}

```

If the hero has five or more arrows, they have plenty and cannot hold any more. The hero has only two arrows, so nothing prints to the console. However, without the braces the logic changes:

```

var arrowsInQuiver = 2
if (arrowsInQuiver >= 5)
    println("Plenty of arrows")
    println("Cannot hold any more arrows")

```

Without the braces, the second **`println`** statement is no longer part of the `if` branch. While "Plenty of arrows" only prints when `arrowsInQuiver` is at least 5, "Cannot hold any more arrows" always prints – no matter

how many arrows the hero is carrying.

For a one-line expression, this overall principle should inform your choice: “Which way of writing the expression is most clear for new readers to understand?” Often, for one-line expressions, removing the curly braces is more readable. For example, removing the curly braces can help to clarify a simple one-line conditional expression like this example:

```
val healthSummary = if (healthPoints != 100) "Need healing!" else "Looking good."
```

By the way, if you are thinking, “OK, but I still don’t love the `if/else` syntax, even with the curly braces. It is just *ugly*.” ... fear not. You are going to rewrite the quest determination expression one last time in a less verbose – and more legible – syntax soon.

Ranges

All the conditions that you wrote in the `if/else` expression for `quest` branch off the value of an integer, `playerLevel`. Most of the branches use the structural equality operator to check whether `playerLevel` is equal to a value, and one uses multiple comparison operators to check whether `playerLevel` is within a range of two numbers. There is a better alternative for the latter: Kotlin provides [ranges](#) to represent a linear series of values.

The [range to operator](#) (`..`) can be used to create a range. A range includes all values from the value on the left of the `..` operator to the value on the right, so `1..5` includes 1, 2, 3, 4, and 5. Ranges can also be a sequence of characters.

You use the `in` operator to check whether a value is within a range. Refactor your `quest` conditional expression to use a range rather than `<=`.

Listing 3.9 Refactoring quest with a range (Main.kt)

```
...
fun main() {
    ...
    val quest: String = if (playerLevel == 1) {
        "Meet Mr. Bubbles in the land of soft things."
    } else if (playerLevel <= 5) {
    } else if (playerLevel in 2..5) {
        // Check whether diplomacy is an option
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")

        if (canTalkToBarbarians) {
            "Convince the barbarians to call off their invasion."
        } else {
            "Save the town from the barbarian invasions."
        }
    } else if (playerLevel == 6) {
        "Locate the enchanted sword."
    } else if (playerLevel == 7) {
        "Recover the long-lost artifact of creation."
    } else if (playerLevel == 8) {
        "Defeat Nogartse, bringer of death and eater of worlds."
    } else {
        "There are no quests right now."
    }
    ...
}
```

Bonus: Using a range in a conditional like this solves the `else if` ordering issue you saw earlier in this chapter. With ranges, your branches can be in any order and the code will evaluate the same.

In addition to the `..` operator, several functions exist for creating ranges. The **`downTo`** function creates a range that descends rather than ascends, for example. And the **`until`** function creates a range that excludes the upper bound of the range specified. You will see some of these functions in a challenge near the end of this chapter, and you will learn more about ranges in [Chapter 9](#).

when Expressions

The when expression is another control flow mechanism available in Kotlin. Like if/else, the when expression allows you to write conditions to check, and it executes corresponding code if the condition evaluates as true. when provides a more concise syntax and is an especially good fit for conditionals with three or more branches.

Suppose that a player can be a member of a fantasy race, like an orc or a gnome, and those fantasy races ally with each other in factions. This when expression takes in a fantasy race and returns the name of the faction to which it belongs:

```
val race = "gnome"
val faction: String = when (race) {
    "dwarf" -> "Keepers of the Mines"
    "gnome" -> "Tinkerers of the Underground"
    "orc", "human" -> "Free People of the Rolling Hills"
    else -> "Shadow Cabal of the Unseen Realm" // Unknown race
}
```

First, a val is declared, `race`. Next, a second val is declared: `faction`, whose value is determined with a when expression. The expression checks the value of `race` against each of the values on the lefthand side of the `->` operator (called the [arrow operator](#)), and when it finds a match it assigns `faction` the value on the righthand side. Multiple cases that have the same output (like orcs and humans) can be placed together, separated by a comma, before the `->`.

(`->` is used differently in other languages – and, in fact, it has other uses in Kotlin, as you will see later in this book.)

By default, a when expression behaves as though there were a `==` equality operator between the argument you provide in parentheses and the conditions you specify in the curly braces. (An *argument* is data that is given to a piece of code as input. You will learn more about arguments in [Chapter 4](#).)

In this example of a when expression, `race` is provided as the argument, so the when expression compares the value of `race` ("gnome") against the first condition ("dwarf") to check whether they are equal. They are not, so the result of the comparison is false, and the when expression moves along to the next condition.

The next comparison is true, so the value in the corresponding branch, "Tinkerers of the Underground", is assigned to `faction`.

Notice that you are using the when expression to assign a value to `faction`. Because the assignment appears outside the when expression, the `faction` assignment will always take place. This means the when expression must always return a value.

When you use a when statement as an expression (such as when you perform an assignment with it), the compiler will require that the when statement be [exhaustive](#), covering every possible input. In this case, without the `else` branch your when statement would not be exhaustive – there are many possible strings that `race` could be set to that are unaccounted for. But the `else` branch adds a fallback option if a rogue value somehow enters your code, so the compiler is satisfied.

Sometimes, a when expression can be exhaustive without an else branch. You will see examples of this in [Chapter 16](#).

Now that you have seen how to use when expressions, you can refine how the `quest` logic is implemented. You previously used an if/else expression, but in this case, a when expression will make your code more readable and concise. A practical rule of thumb is that a when expression should replace an if/else expression if your code includes an `else if` branch.

Update the `quest` logic to use when:

Listing 3.10 Refactoring quest with when (Main.kt)

```
...
fun main() {
    ...
    val quest: String = if (playerLevel == 1) {
        "Meet Mr. Bubbles in the land of soft things."
    val quest: String = when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
    } else if (playerLevel in 2..5) {
        in 2..5 -> {
            // Check whether diplomacy is an option
            val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")

            if (canTalkToBarbarians) {
                "Convince the barbarians to call off their invasion."
            } else {
                "Save the town from the barbarian invasions."
            }
        }
    } else if (playerLevel == 6) {
        "Locate the enchanted sword."
        6 -> "Locate the enchanted sword."
    } else if (playerLevel == 7) {
        "Recover the long-lost artifact of creation."
        7 -> "Recover the long-lost artifact of creation."
    } else if (playerLevel == 8) {
        "Defeat Nogartse, bringer of death and eater of worlds."
    }
}
```

```

        8 -> "Defeat Nogartse, bringer of death and eater of worlds."
    else {
        "There are no quests right now."
        else -> "There are no quests right now."
    }
    ...
}

```

A `when` expression works similarly to an `if/else` expression in that you define conditions and branches that are executed based on a condition being true. `when` is different in that it [scopes](#) the lefthand side of the condition automatically to whatever you provide as an argument to `when`. We will talk about scoping in more depth in [Chapter 4](#) and [Chapter 13](#). For a quick introduction, consider the `in 2..5` branch condition.

You have seen how to use the `in` keyword to check whether a value is within a range, and that is what you are doing here – you are checking the value of `playerLevel`, even though you do not mention it by name. Because the range, on the left of the `->`, is scoped to the `playerLevel` variable, the compiler evaluates `when` expressions as though `playerLevel` were included in each branch condition.

Often, `when` better expresses the logic behind code. In this case, achieving the same result with an `if/else` expression required four `else if` branches. Your `when` expression is much cleaner.

By the way, were you wondering about the nested `if/else` in one branch of your `when` expression? This pattern is not very common, but Kotlin’s `when` expression gives you all the flexibility that you need to implement it.

Run `bounty-board` to confirm that your refactoring of `quest` to use a `when` expression did not change any logic.

when expressions with variable declarations

Sometimes, you will use a `when` expression with an argument that you compute only for the sake of the `when` expression. It can often be handy to use the value of the variable inside one of the conditions of the `when` expression.

For example, suppose you want to assign players a title reflecting their rank, but you only have a `totalExperience` variable of type `Int`. For the sake of simplicity, we will say that it takes 100 experience points to advance between levels (so level 1 means the player has 0-99 experience points, level 2 means the player has 100-199 experience points, and so on). A `when` expression that could tackle this title-generation might look like the following:

```

val playerLevel: Int = totalExperience / 100 + 1
val playerTitle: String = when (playerLevel) {
    1 -> "Apprentice"
    in 2..8 -> "Level " + playerLevel + " Warrior"
    9 -> "Vanquisher of Nogartse"
    else -> "Distinguished Knight"
}

```

This is not bad. However, you can simplify this code even further by moving the variable declaration into the argument of the `when` expression, like this:

```

val playerTitle = when (val playerLevel = totalExperience / 100 + 1) {
    1 -> "Apprentice"
    in 2..8 -> "Level " + playerLevel + " Warrior"
    9 -> "Vanquisher of Nogartse"
    else -> "Distinguished Knight"
}

```

In this `when` expression with a variable declaration, the `playerLevel` `val` exists inside the `when` expression and gets cleaned up when the expression finishes. This lets you avoid having another variable cluttering the rest of your code and lets you use the value without recomputing it every time you need it.

when expressions without arguments

So far, the `when` expressions you have seen have all provided an argument. This has worked well because you have been trying to change your application’s behavior based on a single variable. However, `when` expressions with arguments have some limitations:

- A `when` expression cannot accept multiple arguments.
- Only the `==`, `in`, or `is` operators are allowed in a `when` expression with an argument.

If your conditions involve multiple subjects or you need to use a different comparison operator, you cannot use a `when` expression with an argument. In those cases, you have two options: Use an `if/else` statement, like the ones you saw earlier in this chapter, or use a `when` expression without an argument.

Suppose you wanted to tell the player how many experience points they need to reach the next level in the game. You might have two `Int` variables called `experiencePoints` and `requiredExperiencePoints`. A `when` expression that estimates the amount of experience needed to level up might look like this:

```

val levelUpStatus: String = when {
    experiencePoints > requiredExperiencePoints -> {
        "You already leveled up!"
    }
    experiencePoints == requiredExperiencePoints -> {
        "You have enough experience to level up!"
    }
}

```

```
}
requiredExperiencePoints - experiencePoints < 20 -> {
    // The player needs less than 20 experience points to level up
    "You are very close to leveling up!"
}
else -> "You need more experience to level up!"
}
```

This flexibility means that `if/else` statements and `when` expressions are interchangeable. Any conditions an `if` statement can check for can also appear as conditions for `when` expressions without arguments. You can even use the logical operators shown in [Table 3.2](#).

Challenge: Trying Out Some Ranges

Ranges are a powerful tool in Kotlin, and with some practice you will find the syntax intuitive. For this simple challenge, open the Kotlin REPL (Tools → Kotlin → REPL) and explore some range syntax, including the **toList()**, **downTo**, and **until** functions. Enter the following ranges, one by one. Before pressing Command-Return (Ctrl-Enter) to execute the line and see the result, think about what you expect the result to be.

Listing 3.11 Exploring ranges (REPL)

```
1 in 1..3
(1..3).toList()
1 in 3 downTo 1
1 in 1 until 3
3 in 1 until 3
2 in 1..3
2 !in 1..3
'x' in 'a'..'z'
```

Functions

A [*function*](#) is a reusable portion of code that accomplishes a specific task. Functions are a very important part of programming. In fact, programs are fundamentally a series of functions combined to accomplish more complex tasks.

You have worked with some functions already, like the **println** function, which is provided by the Kotlin standard library for printing data to the console. You can also define your own functions in code that you write. Some functions take in data required to perform a specific task. Some functions also return data, generating output that can be used elsewhere after the function has performed its task.

To get your function feet wet, you will start by using functions to organize bounty-board's existing code. You will then write a second function that further improves your code to read the text on the bounty board.

Extracting Code to Functions

The logic you coded into bounty-board in [Chapter 3](#) is sound, but it would be a better practice to organize it using functions. Your first task is to reorganize your project to encapsulate the logic you have already written to set the text of the bounty board in a function. This will set the stage for adding new features to bounty-board.

Does this mean you are going to delete your code and type the same logic in a different way? Perish the thought. IntelliJ will help you group your logic into a function easily.

Begin by opening your bounty-board project. Make sure the file `Main.kt` is open in the editor.

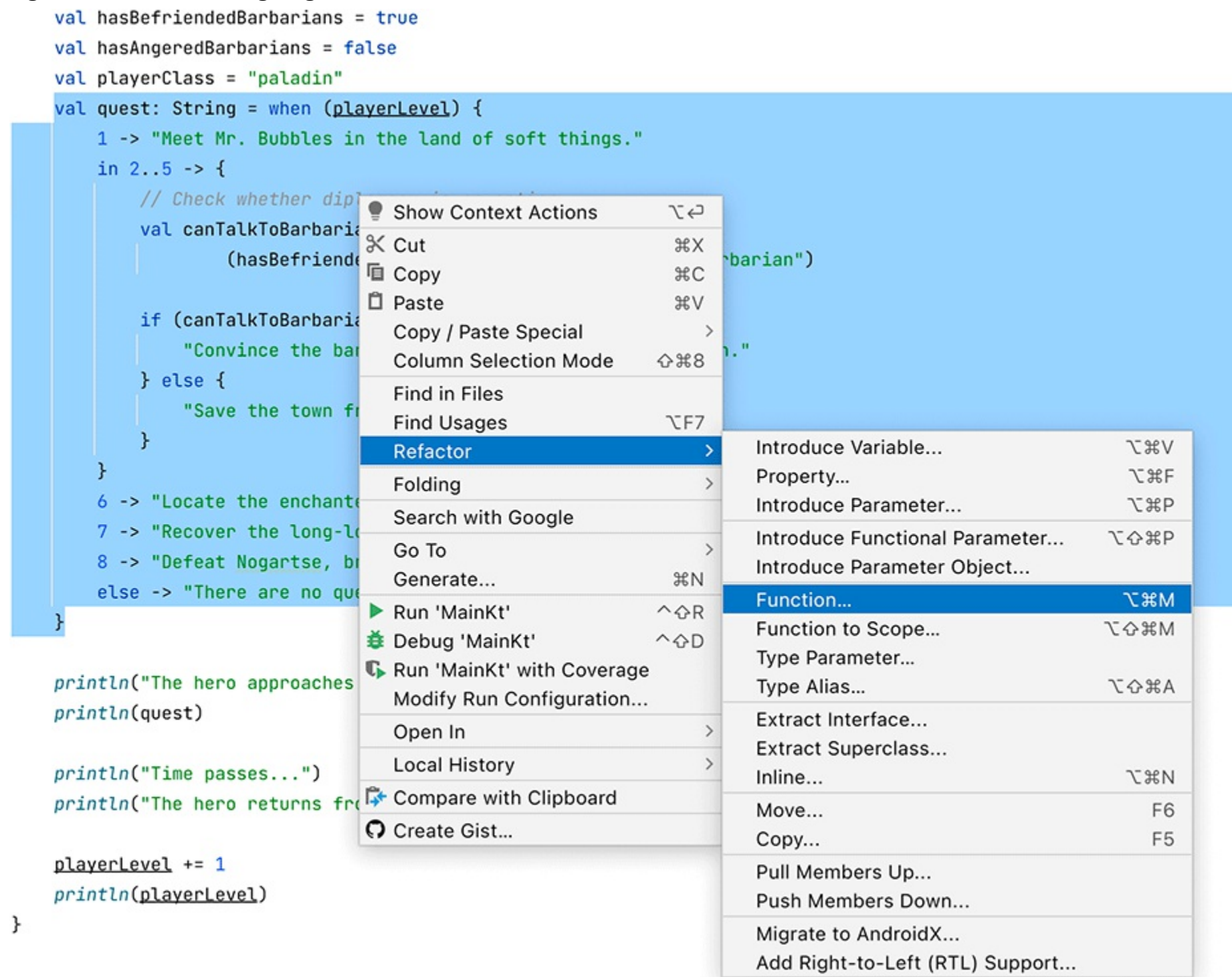
Next, select the conditional code that you defined for generating the `quest` message. Click and drag the cursor, beginning with the line that defines `quest` and ending with the closing curly brace for the `when` expression, like so:

```
...
val quest: String = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    in 2..5 -> {
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")

        if (canTalkToBarbarians) {
            "Convince the barbarians to call off their invasion."
        } else {
            "Save the town from the barbarian invasions."
        }
    }
    6 -> "Locate the enchanted sword."
    7 -> "Recover the long-lost artifact of creation."
    8 -> "Defeat Nogartse, bringer of death and eater of worlds."
    else -> "There are no quests right now."
}
...
```

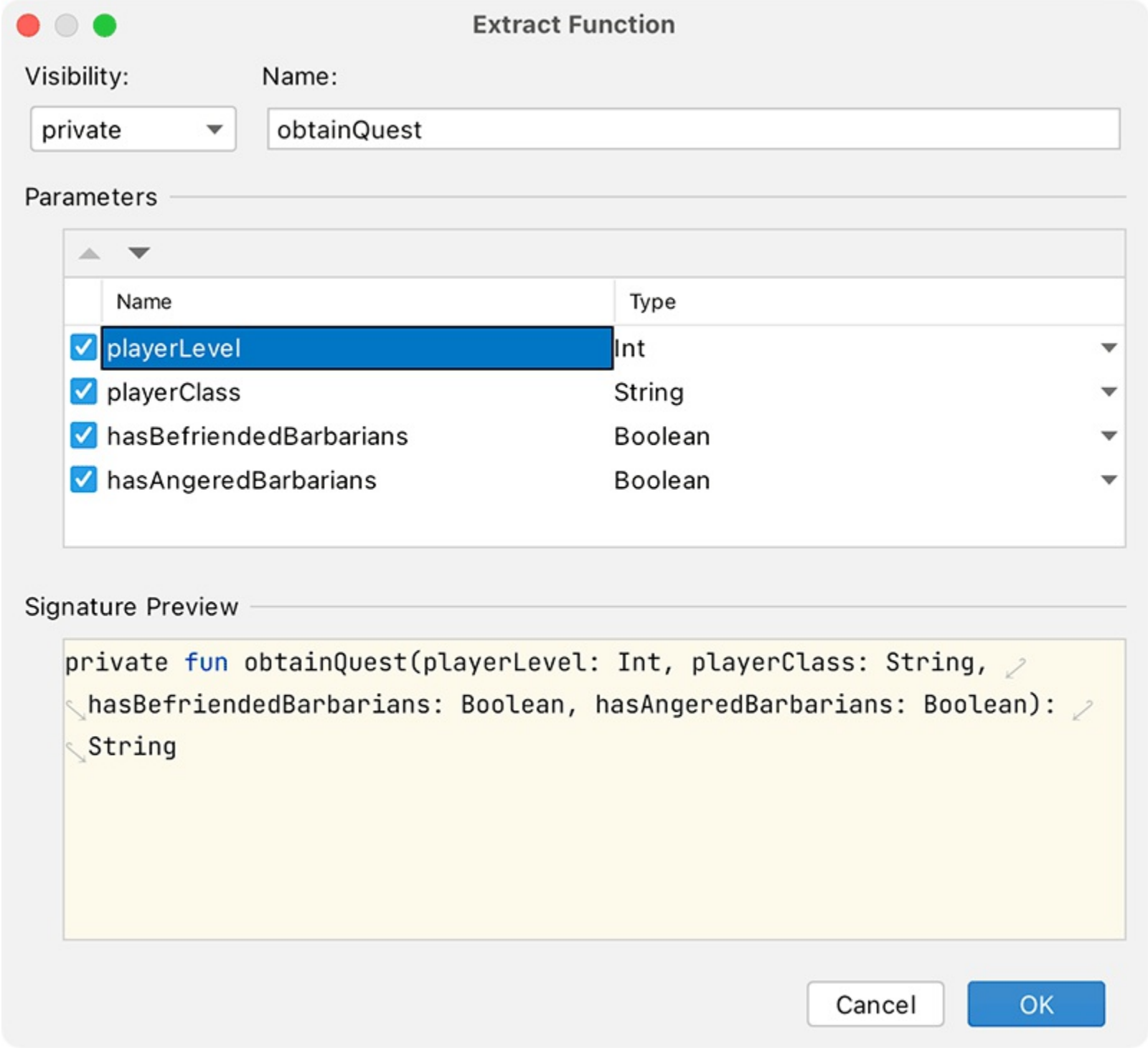
Control-click (right-click) the code you selected and choose Refactor → Function... ([Figure 4.1](#)).

Figure 4.1 Extracting logic to a function



The Extract Function dialog will pop up, as in [Figure 4.2](#):

Figure 4.2 The Extract Function dialog



Enter `obtainQuest` for the name of the function. There are other options in this dialog for the function’s visibility modifier and parameters. We will explain what these options control later in this chapter; for now, make sure they match what is shown in [Figure 4.2](#): The visibility dropdown should be set to `private`, and the parameters should be listed in the order shown – `playerLevel`, `playerClass`, `hasBefriendedBarbarians`, `hasAngeredBarbarians`. You will likely need to reorder the parameters, which you can do by dragging them within the list.

When you are ready, click the `OK` button to proceed. IntelliJ will add a function definition to the bottom of `Main.kt`, like this:

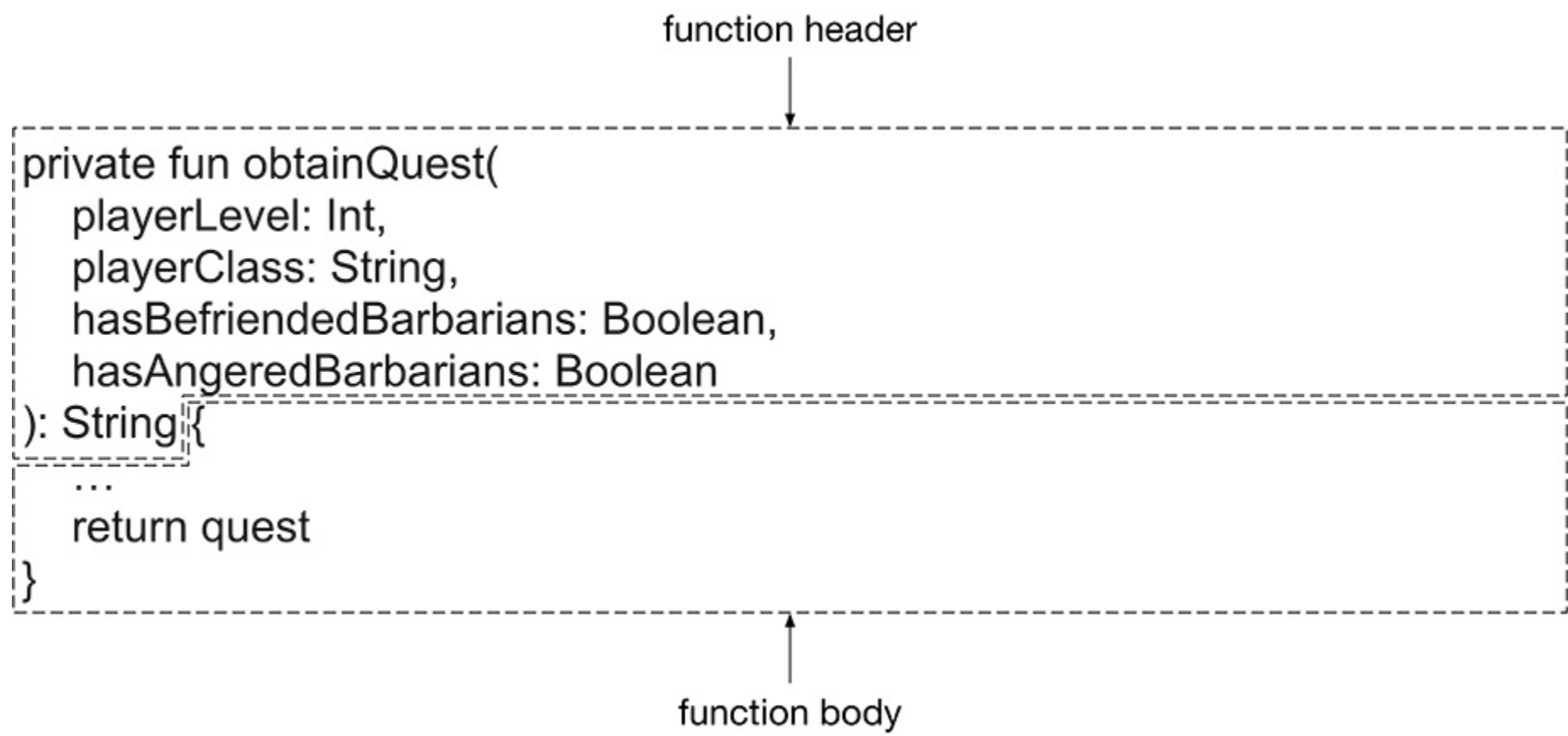
```
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    val quest: String = when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        in 2..5 -> {
            val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")
            if (canTalkToBarbarians) {
                "Convince the barbarians to call off their invasion."
            } else {
                "Save the town from the barbarian invasions."
            }
        }
        6 -> "Locate the enchanted sword."
        7 -> "Recover the long-lost artifact of creation."
        8 -> "Defeat Nogartse, bringer of death and eater of worlds."
        else -> "There are no quests right now."
    }
    return quest
}
```

Your `obtainQuest` function is surrounded by some new code. We will break this down piece by piece next.

Anatomy of a Function

Figure 4.3 shows the two primary parts of a function, the *header* and *body*, using `obtainQuest` as a model:

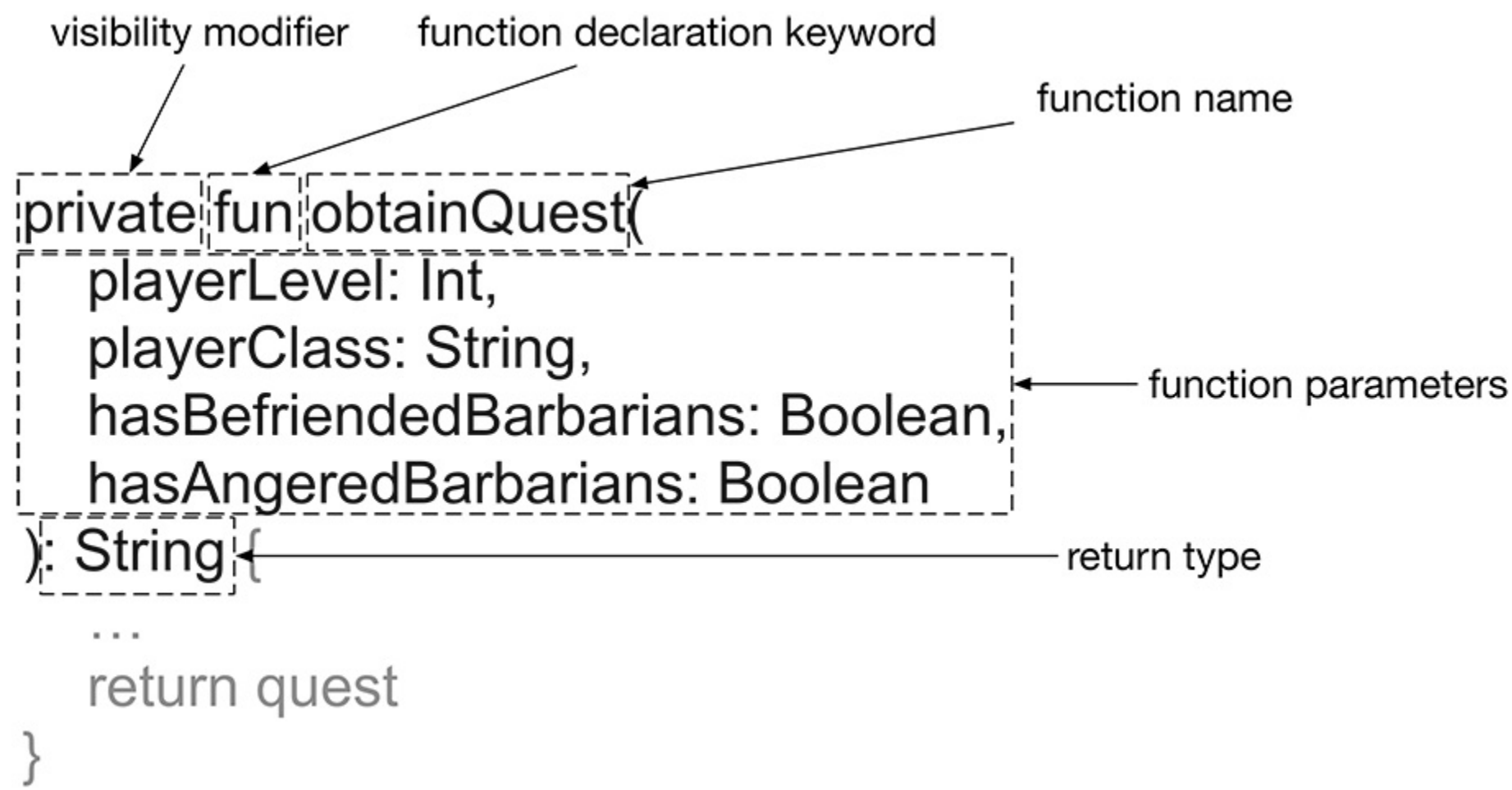
Figure 4.3 A function consists of a function header and a function body



Function header

The first part of a function is the function header. The function header is made up of five parts: the visibility modifier, function declaration keyword, function name, function parameters, and return type (Figure 4.4).

Figure 4.4 Anatomy of a function header



Let's look at each of those elements in some detail.

Visibility modifier

Not all functions should be [visible](#), or accessible, to all other functions. Some might deal with data that should be kept private to a particular file, for example.

A function can optionally begin with a [visibility modifier](#) ([Figure 4.5](#)). The visibility modifier determines which other functions can “see” – and therefore use – the function.

Figure 4.5 Function visibility modifier

```
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    ...
    return quest
}
```

By default, a function’s visibility is public – meaning that all other functions (including functions defined in other files) can use the function. In other words, if you do not specify a modifier for the function, the function is considered public.

In this case, IntelliJ has determined that this function can have private visibility, because the **obtainQuest** function is used only within the current file, `Main.kt`. You will learn more about the available visibility modifiers and how to use them to control which functions can see the function you define in [Chapter 13](#).

Function name declaration

After the visibility modifier (if there is one) comes the `fun` keyword, followed by a name for the function ([Figure 4.6](#)):

Figure 4.6 Function keyword and name declaration

```
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    ...
    return quest
}
```

You specified **obtainQuest** for the function name in the Extract Function dialog, so IntelliJ added `fun obtainQuest` for the function’s name declaration.

Notice that the name **obtainQuest** starts with a lowercase letter and uses camelCase naming with no underscores. All your function names should conform to this official naming convention.

Function parameters

Next come the function parameters ([Figure 4.7](#)):

Figure 4.7 Function parameters

```
private fun obtainQuest(  
    playerLevel: Int,  
    playerClass: String,  
    hasBefriendedBarbarians: Boolean,  
    hasAngeredBarbarians: Boolean  
): String {  
    ...  
    return quest  
}
```

Function [parameters](#) specify the name and type of each input required for the function to perform its task. Functions can require any number of parameters, depending on the task they are designed to perform. Because this function takes many parameters, each one is placed on its own line, following Kotlin's official style. If you had fewer parameters, you could fit the entire function signature on one line. (You will see an example of this when you define your second function.)

For the **obtainQuest** function to determine the quest it should return, the `playerLevel`, `playerClass`, `hasBefriendedBarbarians`, and `hasAngeredBarbarians` variables are all needed, because the `when` expression requires them to check its conditions:

```
private fun obtainQuest(  
    playerLevel: Int,  
    playerClass: String,  
    hasBefriendedBarbarians: Boolean,  
    hasAngeredBarbarians: Boolean  
): String {  
    val quest: String = when (playerLevel) {  
        1 -> "Meet Mr. Bubbles in the land of soft things."  
        in 2..5 -> {  
            val canTalkToBarbarians = !hasAngeredBarbarians &&  
                (hasBefriendedBarbarians || playerClass == "barbarian")  
            ...  
        }  
        ...  
    }  
    return quest  
}
```

Therefore, **obtainQuest**'s function definition specifies that those four variables are required as parameters.

For each parameter, the definition also specifies the type of data it requires. `playerLevel` must be an **Int**, `playerClass` must be a **String**, and `hasBefriendedBarbarians` and `hasAngeredBarbarians` must be **Booleans**.

Note that function parameters are always read-only – they do not support reassignment within the function body. In other words, within the body of a function, a function parameter is a `val`, not a `var`.

Function return type

Many functions generate some type of output; that is their job, to send a value of some type back to where they are called. The final element of the function header is the [return type](#), which defines the type of output that the function will return once it has completed its work. If your function does not need to return any data, you can omit the return type information from the function signature. (Again, you will see this when you define your second function.)

The return type in **obtainQuest** specifies that the function sends back a **String** ([Figure 4.8](#)):

Figure 4.8 Function return type

```
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    ...
    return quest
}
```

Specifically, the function will return one of the `quest` strings, based on the player's level and the values of the other variables.

Function body

After the function header, the function body is defined within curly braces. The body is where the action the function performs is defined. It may also include a return statement that indicates what data to send back.

In this case, the extract function command moved the definition of the `quest` `val` (the code you selected when you ran the command) into the body of the **`obtainQuest`** function.

After that is the new line `return quest`. The `return` keyword indicates to the compiler that the function has finished its work and is ready to return its output data. Here, the output data is `quest`, meaning that the function will return the value of the `quest` variable – the string selected based on the logic in `quest`'s definition.

Function scope

The declaration and assignment for the `quest` variable occur within the function body, and its value is returned at the end of the function body:

```
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    val quest = when (playerLevel) {
        ...
    }
    return quest
}
```

The `quest` variable is referred to as a [local variable](#) because it exists only in the **`obtainQuest`** function's body. Another way to put this is that the `quest` variable exists only within the **`obtainQuest`** function's [scope](#). You can think of scope as the lifespan for a variable.

Because it exists only within the function's scope, `quest` will cease to exist when **`obtainQuest`** completes. The function returns `quest`'s value to its caller, but the variable that held the value is gone when the function completes.

The same is true of the function parameters: The variables `playerLevel`, `playerClass`, `hasBefriendedBarbarians` and `hasAngeredBarbarians` exist within the scope of the function body and cease to exist when the function completes.

In [Chapter 2](#), you saw an example of a variable that was not local to a function or class – a [file-level variable](#):

```
const val HERO_NAME = "Madrigal"

fun main() {
    ...
}
```

Variables like `HERO_NAME` that appear by themselves, outside any function, can be accessed from anywhere in the project (though a visibility modifier can be added to the declaration to change its visibility level). File-level variables remain initialized until program execution stops. By the way, while `HERO_NAME` is a constant, variables

can also be defined at the file level – in fact, you will see an example of this shortly.

Because of the differences between local and file-level variables, the compiler enforces different requirements on when they must be assigned an initial value, or *initialized*.

File-level variables must always be assigned a value when they are defined, or the code will not compile. (Well, almost always. You will see certain exceptions to this in [Chapter 16](#).) This requirement protects you from unexpected – and unwanted – behavior, like a variable not having a value when you try to use it.

Because a local variable is more limited in where it can be used – within the scope of the function in which it is defined – the compiler is more lenient about when it must be initialized. A local variable only has to be initialized before it is used. This means that the following code would be valid:

```
fun main() {  
    var playerLevel: Int  
    println("The hero announces her presence to the world.")  
  
    println(HERO_NAME)  
    playerLevel = 5  
    println(playerLevel)  
    ...  
}
```

As long as you have assigned a value before referencing the variable, the compiler permits the expression.

Calling a Function

IntelliJ not only generated the **obtainQuest** function but also added a line in place of the code it extracted:

```
...
fun main() {
    ...
    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    val quest: String = obtainQuest(playerLevel, playerClass,
                                    hasBefriendedBarbarians, hasAngeredBarbarians)
    ...
}
```

(We have broken this code onto two lines to make it fit the page width. IntelliJ may have placed all of it on one line in your project; this code behaves identically regardless of how many lines it is on.)

This line is a [function call](#), which triggers the function to perform whatever actions are defined in its body. You call a function with its name, along with data to satisfy any parameters required by the function header.

Compare the function header for **obtainQuest** with its corresponding function call:

```
obtainQuest(                // Header
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String

obtainQuest(                // Call
    playerLevel,
    playerClass,
    hasBefriendedBarbarians,
    hasAngeredBarbarians
)
```

The definition of **obtainQuest** shows that it requires four parameters, as discussed above. When you call **obtainQuest**, you include in parentheses the inputs to those parameters. The inputs are called [arguments](#), and providing them to the function is called [passing in arguments](#).

(A note about the terminology: While technically a parameter is what a function requires and an argument is what the caller passes in to fulfill the requirement, you will hear the two terms used interchangeably.)

Here, as the function definition specifies, you pass in the value of `playerLevel` (which, as required, is an **Int**), the **String** value of `playerClass`, and the **Boolean** values of `hasBefriendedBarbarians` and `hasAngeredBarbarians`.

Run bounty-board by clicking the run button, and shazam! You will see the same output you have seen before:

```
The hero announces her presence to the world.
Madrigal
4
The hero approaches the bounty board. It reads:
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

While the output has not changed, bounty-board's code is now more organized and maintainable.

Writing Your Own Functions

Now that you have refactored some of the bounty board’s logic into a function, you can proceed as planned to implement a new function that will read the bounty board. Below your **main** function, define a function called **readBountyBoard** that takes no arguments. Make its visibility private. **readBountyBoard** should have no return statement, but it should print the contents of the bounty board.

readBountyBoard will also need to have access to the `playerLevel` variable. Although you could pass it in as an argument, moving `playerLevel` to a file-level property will make your new function easier to call. When you move the variable declaration, also promote Madrigal from level 4 to level 5 by changing the value of `playerLevel`. This will let you see more interesting quests when you run your code.

Listing 4.1 Adding a readBountyBoard function (Main.kt)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
var playerLevel = 4
    println(playerLevel)

    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    val quest: String = obtainQuest(playerLevel, playerClass,
        hasBefriendedBarbarians, hasAngeredBarbarians)

    println("The hero approaches the bounty board. It reads:")
    println(quest)

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}

private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(obtainQuest(playerLevel, "paladin", true, false))
}
...
```

With your new function in place, you can simplify your **main** function by calling **readBountyBoard** instead of including the full quest determination logic. Replace the existing quest printing logic with a call to **readBountyBoard**. Madrigal should also return to the bounty board once she finishes her quest, so add a second call to your new function at the end of **main**. (**readBountyBoard** was defined without parameters, so you do not pass in any arguments to call it – hence the empty parentheses.)

Listing 4.2 Calling readBountyBoard (Main.kt)

```
...
fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    println(playerLevel)

val hasBefriendedBarbarians = true
val hasAngeredBarbarians = false
val playerClass = "paladin"
val quest: String = obtainQuest(playerLevel, playerClass,
    hasBefriendedBarbarians, hasAngeredBarbarians)

println("The hero approaches the bounty board. It reads:")
println(quest)
    readBountyBoard()

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
    readBountyBoard()
}
...
```

Run bounty-board and admire your new output:

```
The hero announces her presence to the world.
Madrigal
5
The hero approaches the bounty board. It reads:
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
6
The hero approaches the bounty board. It reads:
Locate the enchanted sword.
```

Most of this output has stayed the same. But with the help of your **readBountyBoard** function, it is easier than ever for Madrigal to read quests. It now only takes one line of code to read subsequent quests, which is a great win for code reusability and organization.

Default Arguments

Sometimes an argument for a function has a “usual” value. As an example, it might be very unlikely that Madrigal has angered the barbarians. Depending on her progression in the game, she might not have met the barbarian tribe (if she is just starting out), or such quarrels might be beneath her (if she has advanced to the final boss). Specifying this information for every single call would be redundant. To make it easier to call **obtainQuest**, you can use a [default argument](#).

If a parameter has a default value, then you can call the function without providing a value for that parameter. In that case, the default value will automatically be passed as an argument to the function. Update the **obtainQuest** function with a default value for `hasAngeredBarbarians`:

Listing 4.3 Giving the `hasAngeredBarbarians` parameter a default value (`Main.kt`)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean = false
): String {
    ...
}
```

Now, by default, `hasAngeredBarbarians`’s **Boolean** value will be `false` if no other argument is provided when calling **obtainQuest**. Although this syntax looks similar to the variable declarations you have seen before, there is one subtle difference. Kotlin will not perform type inference on function parameters, so it is always mandatory to specify the type of an argument (even when the type should be obvious to the compiler, such as the Boolean `false`).

Your default argument is now in place, but you are not yet taking advantage of it. Update the **readBountyBoard** function, removing the `false` argument in the call to **obtainQuest**:

Listing 4.4 Using `obtainQuest`’s default argument value (`Main.kt`)

```
...
private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(obtainQuest(playerLevel, "paladin", true, false))
    println(obtainQuest(playerLevel, "paladin", true))
}
...
```

Run `bounty-board` again. Although no argument is specified for `hasAngeredBarbarians`, you will see the same output as before.

Spend another moment updating the `playerClass` and `hasBefriendedBarbarians` parameters to include default values as well.

Listing 4.5 Adding additional default values (`Main.kt`)

```
...
private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(obtainQuest(playerLevel, "paladin", true))
    println(obtainQuest(playerLevel))
}

private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String {
    ...
}
```

Default arguments are a useful tool that allow you to specify optional inputs. Many functions in the Kotlin standard library take advantage of default arguments to change how the function operates.

For example, the **String** type has an **equals** function with a Boolean parameter called `ignoreCase` that specifies whether capitalization matters when checking whether two strings contain the same text. `ignoreCase`’s default value is `false`. This is a great application of default arguments, because capitalization often matters when comparing strings – there is a “default” behavior. Kotlin’s default arguments make it easy to take advantage of behavior that fits most use cases or opt in to a different set of behaviors that a function exposes.

We encourage you to use a default argument when there is a common input for that parameter. For arguments that vary widely when the function is called, or for arguments that have many sensible default values, we recommend not specifying a default argument. Also, it can sometimes be better to require a parameter, in order to make it more discoverable, than to have more concise function calls.

Single-Expression Functions

If you review your code, you might notice that your **main** and **readBountyBoard** functions each have multiple calls to **println**, but **obtainQuest** only does one thing: It produces a quest string based on the player’s status.

In programming, a statement to be evaluated – like a function call or a variable declaration – is called an [expression](#). Currently, **obtainQuest** has two expressions: The declaration and assignment of the **quest** variable, and the return quest statement. You can simplify the function body to use just one expression by removing the intermediate **quest** variable.

Listing 4.6 Removing the intermediate quest variable (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String {
    val quest: String = when (playerLevel) {
    return when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        ...
        else -> "There are no quests right now."
    }
    return quest
}
```

When you have a function with only one expression, a [single-expression function](#), Kotlin allows an alternative syntax: You can omit the return type, curly braces, and return keyword. Make those changes to your **obtainQuest** function:

Listing 4.7 Using optional single-expression function syntax (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
)String {
    return when (playerLevel) {
    = when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        in 2..5 -> {
            val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")

            if (canTalkToBarbarians) {
                "Convince the barbarians to call off their invasion."
            } else {
                "Save the town from the barbarian invasions."
            }
        }
        6 -> "Locate the enchanted sword."
        7 -> "Recover the long-lost artifact of creation."
        8 -> "Defeat Nogartse, bringer of death and eater of worlds."
        else -> "There are no quests right now."
    }
}
```

Instead of using the function body to specify the work the function will perform, with single-expression function syntax you use the assignment operator (=), followed by the expression.

This optional syntax allows you to tighten up the definition for functions with only one expression that is evaluated to perform their task. When you need the results of more than one expression, use the function definition syntax you have already seen.

You also removed the return type information from the function header. Just as Kotlin can infer variable types, it is able to infer the return type of a function when you use the single-expression function syntax you just introduced. You can also manually include the return type, which can improve your code’s readability. In some scenarios, you may be required to include type information for a single-expression function – for example, if the body of the function is inferring its type based on the return type of the function.

Because **obtainQuest** has a complex when expression that spans multiple lines, we recommend including the return type. To add the return type for a single-expression function, insert it between the closing parenthesis ()) and the assignment operator (=).

Listing 4.8 Including return types for single-expression functions (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    ...
}
```

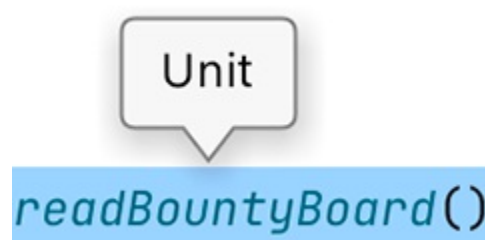
Unit Functions

Not all functions return a value. Some use side effects instead to do their work, like modifying the state of a variable or calling other functions that yield system output. Think about the **readBountyBoard** function and even the **main** function, for example. They define no return type and have no return statement. They use **println** to do their work.

```
private fun readBountyBoard() {  
    println("The hero approaches the bounty board. It reads:")  
    println(obtainQuest(playerLevel))  
}
```

In Kotlin, such functions are known as **Unit** functions, meaning their return type is **Unit**. If you do not specify a return type, Kotlin automatically uses the **Unit** return type. Click one of the **readBountyBoard** function calls and press Control-Shift-P. IntelliJ will display its return type information ([Figure 4.9](#)).

Figure 4.9 **readBountyBoard** is a **Unit** function



What kind of type is **Unit**? Kotlin uses the **Unit** return type to signify exactly this: a function that returns no value. If the return keyword is not used, it is implicit that the return type for that function is **Unit**.

Prior to Kotlin, many languages faced the problem of describing a function that does not return anything. Some languages opted for a keyword `void`, which said, “There is no return type; skip it, because it does not apply.” This seems sound on the surface: If the function does not return anything, skip the type, since there is nothing being returned.

Unfortunately, this solution fails to account for an important feature found in modern languages: generics. Generics are a feature of modern compiled languages that enables a great deal of flexibility. You will investigate generics in Kotlin, which allow you to specify functions that work with many types, in [Chapter 18](#).

What do generics have to do with **Unit** and `void`? Languages that use the `void` keyword have no good way to deal with a generic function that returns nothing. `void` is not a type – in fact, it says, “Type information is not relevant.” And there is no way to describe this “generically,” so these languages miss out on being able to describe generic functions that return nothing.

Kotlin solves this problem by specifying **Unit** for the return type instead. **Unit** indicates a function that does not return anything, but at the same time it is compatible with generic functions that must have some type to work with. This is why Kotlin uses **Unit**: You get the best of both worlds.

Named Function Arguments

Suppose the local village has a blacksmith. You might have a function with the following signature, which could be called to forge some new gear for Madrigal:

```
fun forgeItem(
    itemName: String,
    material: String,
    encrustWithJewels: Boolean = false,
    quantity: Int = 1
): String = ...
```

One way to call this function is by passing arguments for all four parameters:

```
forgeItem("sword", "iron", false, 5)
```

Another way to make the same function call is like this:

```
forgeItem(
    itemName = "sword",
    material = "iron",
    encrustWithJewels = false,
    quantity = 5
)
```

This optional syntax uses [*named function arguments*](#) and is an alternative way to provide arguments to a function. In certain cases, it grants several advantages.

Using named arguments frees you to pass the arguments to the function in any order. For example, you could also call **forgeItem** like this:

```
forgeItem(
    quantity = 5,
    material = "iron",
    itemName = "sword",
    encrustWithJewels = false
)
```

When you do not use named function arguments, you must pass arguments in the order they are defined in the function header.

Another benefit of named function arguments is that they can bring clarity to your code. When a function requires a large number of arguments, keeping track of which argument provides the value for which function parameter can be hard.

This is especially true if the names of the variables passed in do not match the names of the defined function parameters and when there are multiple parameters of the same type. For example, the function call `forgeItem("iron", "sword")` *looks* correct and will compile; you will not realize something has gone wrong until Madrigal is handed an iron made of swords.

Named function arguments always share the name of the parameters they provide values for. This clarity can significantly improve readability, such as when you revisit code that you wrote long ago, when a coworker is reviewing your code, or when you look at a file outside your IDE.

There are also occasions when you will be forced to use named arguments. For example, consider a version of **forgeItem** that does not specify a default argument for `quantity`:

```
fun forgeItem(
    itemName: String,
    material: String,
    encrustWithJewels: Boolean = false,
    quantity: Int
): String = ...
```

If you tried to call this version of **forgeItem** without specifying a value for `encrustWithJewels` and without using named arguments, your code would not compile. The offending function call would look along the lines of: `forgeItem("steel", "dagger", 2)`. For this function call, the compiler thinks that you are providing the value of 2 as the input for `encrustWithJewels`, which is not allowed.

To call this version of **forgeItem** with its default argument, you would have to use a named parameter to make clear which argument the value is for:

```
obtainQuest("steel", "dagger", quantity = 2)
```

Because of this issue, when combining default and named arguments we recommend placing arguments that have default values at the end of the argument list when you declare a function.

It is also possible to use named parameters for only some of the arguments you provide. For example, this is also a valid call to revised version of the **forgeItem** function (with only one default argument for `encrustWithJewels`):

```
forgeItem(
    "gauntlet",
    material = "bronze",
    encrustWithJewels = true,
    1
)
```

If you provide arguments in the same order that they are declared, Kotlin lets you pick and choose which arguments

you used named arguments with. However, as soon as you place a named argument in a different order than it appears in the function's header, you must use named arguments for the remaining arguments.

For example, if you use named arguments to put the `encrustWithJewels` argument before `material`, you would be required to use named arguments below it:

```
forgeItem(  
    "gauntlet",  
    encrustWithJewels = true,  
    material = "bronze",  
    quantity = 1  
)
```

In general, our recommendation is to use named arguments either with every parameter you pass to a function or not at all. If it is fairly well known what the first argument is, you may decide to omit the name for the first argument, but provide names for the second argument onward. We encourage you to experiment in your IDE to see what happens when you use various permutations of named arguments.

In this chapter, you saw how to define functions to encapsulate your code's logic. Your code has become much cleaner and better organized. You also learned a number of the conveniences built into Kotlin's function syntax to enable you to write code that is concise and descriptive: single-expression function syntax, named function arguments, and default arguments.

In the next few chapters, you will use more functions included in Kotlin to improve your bounty-board, starting by taking a deeper dive into Kotlin's numeric types.

For the More Curious: The Nothing Type

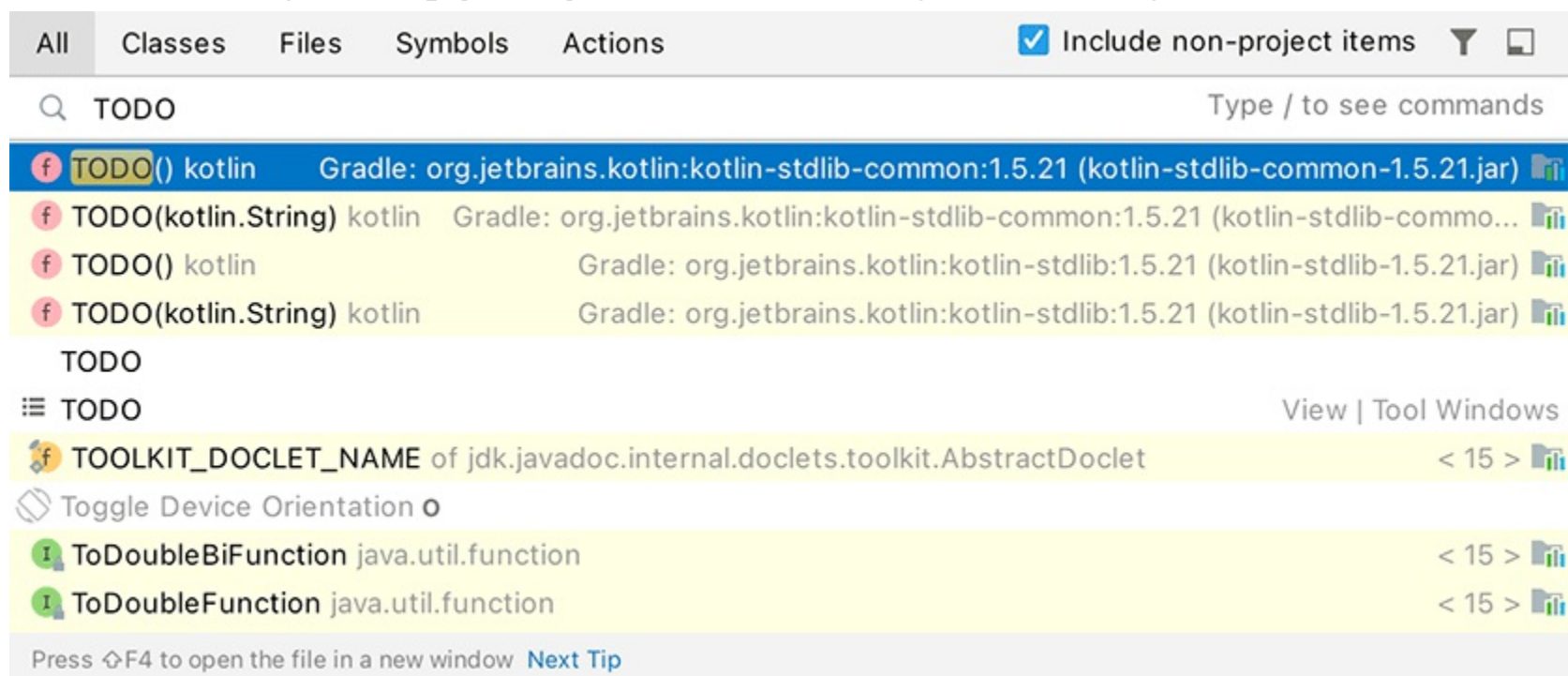
In this chapter you learned about the **Unit** type and that a function of the **Unit** type returns no value.

Another type related to **Unit** is the **Nothing** type. Like **Unit**, **Nothing** indicates that a function returns no value – but there the similarity ends. **Nothing** lets the compiler know that a function is guaranteed to never successfully complete; the function will either throw an exception or for some other reason never return to where it was called.

What is the use of the **Nothing** type? One example of **Nothing**'s use is the **TODO** function, included with the Kotlin standard library.

Take a look at **TODO**. You can search for it by pressing the Shift key twice to open the Search Everywhere dialog. Check the Include non-project items box at the top and enter `TODO` in the search field. Select `TODO() (kotlin)` from the list of results, as shown in [Figure 4.10](#).

Figure 4.10 Selecting `TODO() (kotlin)` in the Search Everywhere dialog



You will see this declaration:

```
/**
 * Always throws [NotImplementedError] stating that operation is not implemented.
 */
public inline fun TODO(): Nothing = throw NotImplementedError()
```

TODO throws an exception – in other words, it is guaranteed to never complete successfully – and returns the **Nothing** type.

When would you use **TODO**? The answer is in the name: It tells you what you still have “to do.” Consider a function that has yet to be implemented, and instead calls **TODO**:

```
fun shouldReturnAString(): String {
    TODO("implement the string building functionality here to return a string")
}
```

The developer knows that the **shouldReturnAString** function should return a **String**, but they have not yet completed the other features needed to implement it. The return type for **shouldReturnAString** is a **String**, but the function never actually returns anything at all. Because of **TODO**'s return value, that is perfectly fine.

TODO's **Nothing** return type indicates to the compiler that the function will never complete successfully, so checking the return type in the function body is not required past **TODO** because **shouldReturnAString** will never return. The compiler is happy, and the developer is able to continue feature development without completing the implementation for **shouldReturnAString** until all the details are ready.

Another feature of **Nothing** that is useful in development is that if you add code below the **TODO** function, the compiler will show a warning indicating that the code is unreachable ([Figure 4.11](#)):

Figure 4.11 Unreachable code

```
fun shouldReturnAString(): String {  
    TODO()  
    println("This is unreachable")  
}
```

Unreachable code

[kotlin.io](#) *ConsoleKt.class*

[@InlineOnly](#)

```
public inline fun println(  
    message: Any?  
): Unit
```

Prints the given [message](#) and the line separator to the standard output stream.

 Gradle:

org.jetbrains.kotlin:kotlin-stdlib:1.5.21



Because of the **Nothing** type, the compiler can make this assertion: It is aware that **TODO** will not successfully complete; therefore, all code after **TODO** is unreachable.

For the More Curious: File-Level Functions in Java

All the functions you have written so far have been defined at the file level in `Main.kt`. If you are a Java developer, then this may seem surprising to you. In Java, functions and variables can only be defined within classes, but this is a rule that Kotlin does not adhere to.

How is this possible if Kotlin code compiles to Java bytecode to run on the JVM? Does Kotlin not have to play by the same rules? A look at the decompiled Java bytecode for `Main.kt` should prove illuminating:

```
public final class MainKt {
    ...

    public static final void main() {
        ...
    }

    private static final void readBountyBoard() {
        ...
    }

    private static final String obtainQuest(...) {
        ...
    }

    // $FF: synthetic method
    static String obtainQuest$default(...) {
        ...
    }
}
```

File-level functions are represented in Java as static methods on a class with a name based on the file in which they are declared in Kotlin. ([Method](#) is Java for *function*.) In this case, functions and variables defined in `Main.kt` are defined in Java in a class called **MainKt**.

You will see how to declare functions within classes in [Chapter 13](#), but being able to declare functions and variables outside them gives you more flexibility to define a function that is not tied to a particular class definition. (And if you are wondering what the **obtainQuest\$default** method in **MainKt** is all about, this is how default arguments are implemented. You will see this in more detail in [Chapter 23](#).)

For the More Curious: Function Overloading

The **obtainQuest** function you defined, with its default arguments for the `playerClass`, `hasBefriendedBarbarians`, and `hasAngeredBarbarians` parameters, can be called in many ways. To list a few (without using named arguments):

```
obtainQuest(playerLevel)
obtainQuest(playerLevel, playerClass)
obtainQuest(playerLevel, playerClass, hasBefriendedBarbarians)
obtainQuest(playerLevel, playerClass, hasBefriendedBarbarians,
            hasAngeredBarbarians)
```

When a function has multiple signatures, like **obtainQuest**, it is said to be [overloaded](#). Overloading is not always the result of a default argument. You can also define multiple implementations with the same function name. To see what this looks like, open the Kotlin REPL (Tools → Kotlin → Kotlin REPL) and enter these function definitions:

Listing 4.9 Defining an overloaded function (REPL)

```
fun performCombat() {
    println("You see nothing to fight!")
}

fun performCombat(enemyName: String) {
    println("You begin fighting $enemyName.")
}

fun performCombat(enemyName: String, isBlessed: Boolean) {
    val combatMessage: String = if (isBlessed) {
        "You begin fighting $enemyName. You are blessed with 2X damage!"
    } else {
        "You begin fighting $enemyName."
    }
    println(combatMessage)
}
```

You have defined three implementations of **performCombat**. All are **Unit** functions, with no return value. One takes no arguments. One takes a single argument, the name of an enemy. The last takes two arguments: the enemy's name and a Boolean indicating whether the player is blessed. Each function generates a different message (or messages) through calls to **println**.

When you call **performCombat**, how will the REPL know which one you want? It will evaluate the arguments you pass in and find the implementation that matches the number and type of the arguments. In the REPL, call each of the implementations of **performCombat**, as shown. (Remember to press Command-Return [Ctrl-Enter] to execute the code.)

Listing 4.10 Calling the overloaded functions (REPL)

```
performCombat()
performCombat("Ulrich")
performCombat("Hildr", true)
```

Your output will read:

```
You see nothing to fight!
You begin fighting Ulrich.
You begin fighting Hildr. You are blessed with 2X damage!
```

The implementation of the overloaded function was selected based on how many arguments you provided.

For the More Curious: Function Names in Backticks

Kotlin includes a feature that might, at first glance, seem slightly peculiar: the ability to define or invoke a function named using spaces and other unusual characters, so long as they are surrounded using the backtick symbol, ```. For example, you can define a function like this:

```
fun `**~prolly not a good idea!~**`() {  
    ...  
}
```

And you could then invoke ``**~prolly not a good idea!~**`` like this:

```
`**~prolly not a good idea!~**`()
```

Why is this feature included? You should never name a function anything like ``**~prolly not a good idea!~**``. (Nor with an emoji. Please backtick responsibly.) There are several valid reasons the function name backticks exist.

The first is to support interoperability with other languages. Kotlin includes great support for invoking methods from existing Java, C, and JavaScript code within a Kotlin file. (You will tour a number of interoperability features in [Part VI](#).) Because Kotlin has different [reserved keywords](#) (words that are forbidden for use as function names) than the languages it targets, the function name backticks allow you to dodge any potential conflict when interoperability is important.

For example, imagine a Java method name from a legacy Java project, **is**:

```
public static void is() {  
    ...  
}
```

In Kotlin, `is` is a reserved keyword (the Kotlin language includes an `is` keyword; it allows you to check the type of an instance, as you will see in [Chapter 15](#)). But in other languages, **is** is a valid method name. With backticks, you are able to invoke the **is** method from Kotlin, like so:

```
fun doStuff() {  
    `is`() // Invokes the Java `is` method from Kotlin  
}
```

In this case, the backtick feature supports interoperating with a Java method that would otherwise be inaccessible due to its name.

The second reason for the feature is to support more expressive names of functions that are used in a testing file. For example, a function name like this:

```
fun `users should be signed out when they click logout`() {  
    // Do test  
}
```

is more expressive and readable than this:

```
fun usersShouldBeSignedOutWhenTheyClickLogout() {  
    // Do test  
}
```

Using backticks to provide an expressive name for a test function is the exception to the “lowercase first letter, followed by camelCase” naming standard for functions.

5 Numbers

Kotlin has a variety of types for dealing with numbers and numeric computations, including multiple types for both whole-number integers and numbers with decimals (also called [*floating point numbers*](#)).

In this chapter, you will see how Kotlin handles both varieties. You will not be making any changes to bounty-board in this chapter; instead, you will be evaluating code using the REPL. Keep your bounty-board project around, though, because you will be working on it again in the next chapter.

Numeric Types

Kotlin supports several numeric types. Regardless of which platform you target, the rules for these numeric types stay the same. As of Kotlin 1.5, numeric types in Kotlin come in two flavors: *signed* and *unsigned*. Signed numbers can represent both positive and negative numbers. Unsigned numbers can only represent positive numbers. We will discuss signed numbers first and come back to their unsigned counterparts in [the section called *For the More Curious: Unsigned Numbers*](#) near the end of this chapter.

In addition to whether a number has a sign, there are several key differences between Kotlin’s numeric types. Numeric types primarily differ in the number of bits they are allocated in memory and, consequently, their minimum and maximum values.

If you are familiar with Java, these rules may look very familiar. Kotlin uses the same rules as Java for its numeric types. For those coming from JavaScript, you may be surprised to see how many types of numbers there are, since JavaScript has a single **Number** type. Each of these types has its own meaning.

Last, for those intending to use Kotlin/Native, pay attention to the number of bits each type is allocated. Kotlin’s numeric types behave identically and have the same number of bits anywhere you use the language – regardless of platform (as opposed to C’s `int` type, which can have a different number of bits depending on how the program is compiled).

[Table 5.1](#) shows some of the numeric types in Kotlin, the number of bits for each type, and the maximum and minimum values the types support.

Table 5.1 Commonly used numeric types

Type	Bits	Max Value	Min Value
Byte	8	127	-128
Short	16	32,767	-32,768
Int	32	2,147,483,647	-2,147,483,648
Long	64	9,223,372,036,854,775,807	-9,223,372,036,854,775,808
Float	32	3.4028235E38	1.4E-45
Double	64	1.7976931348623157E308	4.9E-324

There is a relationship between a type’s bit size and its maximum and minimum values. Computers store integers in binary form with a fixed number of bits (“bit” is short for “binary digit,” by the way). A bit is represented by a single 0 or 1.

To represent a number, Kotlin assigns a finite number of bits, depending on the numeric type chosen. For signed numbers, the leftmost bit position represents the sign (0 for positive, 1 for negative). The remaining bit positions each represent a power of 2, with the rightmost position being 2^0 . To compute the value of a binary number, add up each of the powers of 2 whose bit is a 1.

[Figure 5.1](#) shows the example of the number 42 in binary form.

Figure 5.1 42 in binary

1

0

1

0

1

0

$= 2^5 + 2^3 + 2^1 = 32 + 8 + 2 = 42$

2^5

2^4

2^3

2^2

2^1

2^0

Since **Int** is 32 bit, the largest number that can be stored in an **Int** is represented, in its binary form, with 31 ones. (Remember, the leftmost bit represents the sign.) Adding up all those powers of 2 yields a total of 2,147,483,647, the largest value an **Int** in Kotlin can hold.

Because the number of bits determines the maximum and minimum values a numeric type can represent, the difference between the types is the number of bits available to represent the number. Since **Long** uses 64 bits instead of 32, a **Long** can hold an exponentially larger number (2^{63}).

A note about the types **Short** and **Byte**. The long and short of it (sorry) is that neither **Short** nor **Byte** is commonly used when representing conventional numbers. They are used for specialized cases and to support interoperability, typically with legacy programs.

For example, you might work with **Byte** when reading a stream of data from a file or processing graphics (a color

pixel is often represented as three bytes: one for each color in RGB). You will sometimes see **Short** used when interacting with native code for CPUs that do not support 32-bit instructions. However, for most purposes, whole numbers are represented with **Int** or, when a greater value is needed, **Long**.

Integers

You learned in [Chapter 2](#) that an integer is a number that does not have a decimal point – a whole number – and can be represented in Kotlin with the **Int** type. **Int** is good for representing a quantity or count of “things”: the player’s skill level, experience points, the remaining pints of mead, or the count of gold and silver coins a player possesses.

To get some more firsthand experience with the Int type, you will be using the REPL to perform some arithmetic operations. Open the Kotlin REPL by clicking Tools → Kotlin → Kotlin REPL.

In the REPL, enter the following operation. What do you think the result will be?

Listing 5.1 Performing integer arithmetic (REPL)

```
2 + 4 * 5
```

Run this expression by pressing Command-Return (Ctrl-Enter). The REPL will print the result 22. Was your guess correct?

As this example shows, Kotlin’s multiplicative operators (*, /, and %) take precedence over additive operators (+, -), as they do in regular mathematics. This means that the multiplication of 4 * 5 gets evaluated before adding 2 to the value.

If you need to specify a different order (or if you want to make the order more obvious), you can group operations in parentheses. As you saw in [Chapter 3](#), expressions nested in parentheses are evaluated first.

Now enter the following operation in the REPL. Again, what would you guess the result of this expression will be?

Listing 5.2 Performing integer division (REPL)

```
9 / 5
```

Evaluate the expression. You might be expecting a result of 1.8, but the REPL will print a result of 1. Why did this happen?

When you divide an integer by another integer, the result will always be an integer. If the result of an integer division operation is not a whole number, Kotlin will truncate all the digits after the decimal point. Similarly, if you ask the REPL to evaluate 9 / -5, the result will be -1.

The result of an integer division will always be rounded toward zero. This truncation happens silently, so you need to be careful when performing integer division if the remainder is important to your application.

One way to calculate the remainder of an operation is to use the [modulus operator](#) (%), also known as the [remainder operator](#), which finds the remainder when one number is divided by another. For example, 9 % 5 will return a result of 4.

If your numbers are not whole numbers, integers will not be your friend. For decimal numbers, look instead to the floating point number types.

Floating Point Numbers

In Kotlin, **Float** and **Double** are the two numeric types that can represent decimal numbers. These numbers are also called “floating point numbers” because the decimal point can appear in any position – that is, the decimal point is “floating” (as opposed to being fixed) – depending on the number’s order of magnitude.

The name **Double** is short for “double-precision floating point number.” Doubles use twice as many bits as a regular **Float**, hence the name, and can store decimal numbers more accurately.

Floating point numbers in Kotlin can also take on special values for infinity, negative infinity, and NaN – short for “not a number.” These values are typically returned when performing illegal or undefined operations, like division by zero (which returns either infinity or negative infinity) or the square root of a negative number (which returns NaN). You can access these special values by referencing `Double.POSITIVE_INFINITY` (or `Float.POSITIVE_INFINITY`), `Double.NEGATIVE_INFINITY` (or `Float.NEGATIVE_INFINITY`), and `Double.NaN` (or `Float.NaN`) in your code.

In the REPL, revisit your `9 / 5` integer division expression. To make this expression use floating point division, you need to tell Kotlin that these numbers are floating point numbers instead of integers. One way to do that is to write the numbers with decimal points.

Listing 5.3 Performing floating point division (REPL)

```
9.0 / 5.0
```

Evaluate this expression. The REPL will print out `kotlin.Double = 1.8`, which is more in line with what we as humans would expect. Note that the type of this expression is a **Double**. Kotlin by default prefers doubles, but you can specifically request that your value be treated as a **Float** by adding the `f` suffix to your numbers: `9.0f / 5.0f`.

(If you specify that the number is a float with the `f` suffix, you can omit the `.0` from both numbers, if you prefer. In fact, you could even express the same operation as `9f / 5`, because Kotlin will use floating point division if at least one of the operands is a floating point number.)

We mentioned earlier that floating point numbers have a “precision.” To see what we mean, type this expression into the REPL:

Listing 5.4 Causing a floating point precision error (REPL)

```
0.01f * 5
```

Intuitively, this expression should return 0.05. However, when you evaluate the expression, the REPL will happily print a result of 0.049999997. Now compare whether this result is equal to 0.05:

Listing 5.5 Checking exact floating point equality (REPL)

```
0.01f * 5 == 0.05f
```

When you execute this statement, the REPL will print the result `false`. Why?

When you are working with integers, each bit has a specific meaning that never changes. With floating point numbers, it is not so simple. At a binary level, floating point values are composed of one bit for the sign and two additional sets of bits: The first set represents an exponent that determines the order of magnitude of the number. The second set of bits determines the significant digits in the number being represented.

A consequence of this is that floating point numbers cannot precisely represent every number – they are *approximations*. While 0.05 can be precisely represented using a **Float**, 0.01 cannot. The closest that a **Float** can get to storing 0.01 is 0.009999999776482582. When multiplying `0.01f * 5`, the loss of precision propagates to the result, leaving you with an inaccurate (but close!) result.

To avoid this issue of precision, you have several options:

- *Prefer **Double** instead of **Float**:* By using a higher-precision floating point type, you are less likely to encounter floating point precision errors – at the expense of higher memory use. Be aware that using a **Double** is still not enough to completely avoid this issue, though. (Try evaluating `10.1 - 5.9` in the REPL.)
- *Round floating point values:* If you know exactly how many decimal points a floating point value should have, you can round it accordingly. Kotlin has access to APIs that can print decimal values with a specific number of decimal points, as well as a **round** function that will round numbers to the closest whole number.

If you combine the round function with some multiplication and division, you can round to a specific number of decimal points. For example, `round(number * 100) / 100` would round `number` to two decimal places.

- *Prefer another data type with higher precision:* If your application needs to store a mission-critical decimal value where rounding and loss of precision are unacceptable (for example, if you are working on banking software), you may need a more robust data type.

Sometimes, you can use an **Int** to represent data types that you think of as being decimal. For example, if you want to store a user's bank account balance, you could use an **Int** to track the value in cents rather than a **Double** that tracks the value in dollars.

As a last resort, **BigDecimal** is a class available to you when targeting the JVM. **BigDecimal** has much more power around how rounding and operations are performed, and it sidesteps precision errors at the expense of increased complexity compared to the basic numeric types. **BigDecimal** also uses more resources than a floating point number to store values and perform arithmetic operations. (**Decimal** is the equivalent class of **BigDecimal** if your Kotlin code is targeting iOS or macOS.)

Formatting a Double

Think about the fantasy game world of bounty-board. Suppose you wanted to track the amount of Madrigal’s in-game currency. You might have logic that looks like this:

```
val currentBalance = 1120.40
println(currentBalance)
```

If you run this code, Madrigal’s bank account balance would be printed out as 1120.4, with no additional formatting.

Rather than working with 1120.4 arbitrary units of money, it would be better if this balance was formatted more like currency. For North American countries, this means the balance would display as \$1,120.40. **String**’s **format** function can be used to apply formatting to a double, including a currency or other symbol, a thousands separator, and the number of decimal places shown.

Start with the number of decimal places. Run this code in the REPL, paying attention to the use of the format function.

Listing 5.6 Formatting a double (REPL)

```
val currentBalance = 1120.40
println("%.2f".format(currentBalance))
```

(Here you have used dot syntax to invoke the **format** function. This syntax is used any time you invoke a function that is included as part of a type’s definition.)

The REPL prints 1120.40, which is already more readable.

The call to **format** specifies a *format string*, "%.2f". A format string uses a special sequence of characters to define how you want to format data. The particular format string you defined specifies that you want to round the floating point number up to the second decimal place. Then you pass the value or values to format as an argument to the **format** function.

These format strings use the same style as the standard format strings in Java, C/C++, Ruby, and many other languages. For details on format string specification, take a look at the Java API documentation at docs.oracle.com/javase/8/docs/api/java/util/Formatter.html.

To add the comma and the dollar sign, tweak the format to "\$%, .2f":

Listing 5.7 Adding currency formatting (REPL)

```
val currentBalance = 1120.40
println("$%, .2f".format(currentBalance))
```

There are a few caveats to using **format**. The first is that it can make your application harder to localize. If you wanted Madrigal’s balance to be shown in the user’s locale, then you would need to replace the dollar sign with the appropriate currency symbol. Also, many countries use a comma as a decimal point and periods as thousands separators, which adds more complexity to your localization efforts.

The second caveat is that (at time of this writing) **format** is only available when targeting the JVM. If you need to target other platforms, you will need a different approach to formatting numbers.

To address both of these caveats, you can use a platform-specific formatting API. On Java, you have access to the **NumberFormat** class. To achieve the same result using **NumberFormat**, you use code like this:

```
val currentBalance = 1120.40
val formatter = NumberFormat.getCurrencyInstance()
val formattedBalance = formatter.format(currentBalance)
println("Madrigal's life savings: " + formattedBalance)
```

(To run this code in the REPL, you would need to begin with the line `import java.text.NumberFormat` to import the class.)

This would automatically convert Madrigal’s savings amount into a formatted string suitable for the user based on their system-wide locale preferences.

Android has its own **NumberFormat** class in the `android.icu.text` package in addition to the Java flavor. Both **NumberFormat** classes can be used to obtain instances like locale-appropriate currency formats. Similarly, if your Kotlin code targets iOS or macOS, you have access to **NSNumberFormatter**. For Kotlin/JS, meanwhile, you can use **Intl.NumberFormat** for your number formatting needs.

You will see how to use both **NSNumberFormatter** and **Intl.NumberFormat** with Kotlin/Native and Kotlin/JS in [Part VI](#) of this book.

Converting Between Numeric Types

You will sometimes need to convert between floating point numbers and integer numbers. For example, suppose you kept track of a player's skill level using experience points instead of tracking the level of the player directly, like you do in bounty-board. You might have code that looks like this:

```
var experiencePoints = 460.25
val playerLevel = experiencePoints / 100
```

What type do you expect that `playerLevel` will be? (You can type this code into the REPL, select `playerLevel`, and press Control-Shift-P [Ctrl-Shift-P] to see the answer for yourself.)

As you may have guessed, `playerLevel`'s type will be inferred as **Double**, and its value will be set to 4.6025 when you run this code. This is probably not what you want. For the sake of determining quest difficulty, a level 4 character would be at level 4 whether they have 400 experience points or 499 experience points. To account for this detail, `playerLevel` should be an **Int**, not a **Double**.

To make this conversion, you can call **toInt** on the expression. Give it a try in the REPL:

Listing 5.8 Converting a Double to an Int (REPL)

```
var experiencePoints = 460.25
val playerLevel = (experiencePoints / 100).toInt()
println(playerLevel)
```

When you evaluate this code, the REPL will print out the result 4. When you convert a **Double** to an **Int** in this way, it follows the same rules you saw when performing integer division: Values after the decimal point get truncated, and the number gets rounded toward zero.

This behavior is sometimes referred to as *loss of precision*. Some portion of the original data is lost, because you asked for an integer representation of a double that included a fractional quantity, and the integer representation is less precise.

Sometimes, this truncating behavior is undesirable. There are occasions when you will want a **Double** to be *rounded* to an **Int** instead of converted. Luckily, Kotlin also has a **roundToInt** function, which will do just that. Enter the code in [Listing 5.9](#) into the REPL to see this function in action.

Listing 5.9 Rounding a Double to an Int (REPL)

```
import kotlin.math.roundToInt
val distanceToObjective = 4.6
println("The objective is about " + distanceToObjective.roundToInt() + " miles away")
```

When you run this code, you will see the output The objective is about 5 miles away. Kotlin prints 5 instead of 5.0, which indicates that the value is an integer and not a floating point number. This function rounds and converts the **Double** to an **Int** all in one step. This makes **roundToInt** a helpful alternative to **toInt**, depending on your needs.

(By the way, if you need to convert an **Int** into a **Double** you can use the corresponding **toDouble** function. When you call this function, it will return a double with zeros after the decimal point.)

In this chapter, you have worked with Kotlin's numeric types and learned how Kotlin handles the two major categories of numbers: whole numbers and decimal point numbers. You have also learned how to convert between the different types and what each type supports. In the next chapter, you will learn about Kotlin's strings.

For the More Curious: Unsigned Numbers

Kotlin 1.5 introduced unsigned numeric types as a stable language feature. They are very similar to the numeric types you have seen so far, except that they cannot express negative values. [Table 5.2](#) shows the unsigned numeric types available in Kotlin.

Table 5.2 Unsigned numeric types in Kotlin

Type	Bits	Max Value	Min Value
UByte	8	255	0
UShort	16	65,535	0
UInt	32	4,294,967,295	0
ULong	64	18,446,744,073,709,551,615	0

There are similarities and differences between these unsigned numeric types and the signed numeric types introduced earlier in this chapter. For one thing, the floating-point numeric types – **Float** and **Double** – have no unsigned counterparts. We will come back to why this is at the end of this section.

Each of the integer numeric types – **Byte**, **Short**, **Int**, and **Long** – does have an equivalent unsigned type – **UByte**, **UShort**, **UInt**, and **ULong**. Signed and unsigned numbers of these types have the same number of bits; for example, an **Int** and a **UInt** each has 32 bits. However, the unsigned versions all have a minimum value of 0 and a much higher maximum value than their signed counterparts.

Using an unsigned numeric type takes a bit more effort than a signed number. Declaring `playerLevel` as a **UInt** and assigning it a value of 5 looks like this:

```
var playerLevel: UInt = 5.toUInt()
```

You can also suffix a number literal with the letter `u` to mark it as unsigned, like this:

```
var playerLevel: UInt = 5u
```

You can remove the explicit type information (`: UInt`) if you want, but you must use either the unsigned suffix or the **toUInt** function call to mark the number as an unsigned value.

Kotlin will not implicitly convert between signed and unsigned types. This also affects how you perform operations on an instance of an unsigned type. For example:

```
var playerLevel = 5u
val levelsToAdd = 1
playerLevel += 1.toUInt() // Adding a UInt to a UInt is allowed
playerLevel += 1u        // Also allowed (shorthand for the line above)
playerLevel += 1          // Compiler error: you must convert 1 to a UInt first
playerLevel += levelsToAdd // Compiler error: cannot add an Int and a UInt

print(playerLevel * 10u) // Allowed
print(playerLevel * 10)  // Compiler error
```

Because Kotlin does not automatically convert between signed and unsigned types, you will need to either make many variables in your program unsigned or utilize the **toUInt** and **toInt** functions to convert back and forth as necessary.

Unsigned numbers can be useful in specific use cases, like when you need to guarantee a variable is positive, enforce rules on function arguments, or if you are working with platform-specific code that uses an unsigned number. But unsigned numbers are not bulletproof. It is still possible to set up unexpected scenarios in your program if you are not careful. For example, take a look at what happens if you use a **UInt** to hold the `playerLevel` and assign it to -1:

```
val playerLevel = (-1).toUInt()
println(playerLevel) // Prints 4294967295
```

If you subtract 1 from the lowest value any numeric type can express, it “rolls over” to the largest value the type can represent. This is known as *integer underflow*. In this case, the lowest value that **UInt** or any unsigned type can express is 0, so subtracting 1 results in the highest possible **UInt** value: 4,294,967,295. This unexpected result is one of the pitfalls of working with signed and unsigned types together.

(By the way, the reverse can also happen and is called *integer overflow*. This usually surfaces when you are working with very big signed numbers – if you add 1 to `Int.MAX_VALUE`, you wrap around and are given `Int.MIN_VALUE`.)

You might be wondering why unsigned numbers behave differently than the signed numeric types you saw throughout this chapter. Why so many hoops to jump through? In Kotlin, unsigned integers are not implemented in the same way as signed integers. In fact, unsigned integers are implemented *with* signed integers.

When you use an unsigned integer, you are actually telling Kotlin to use a signed integer but interpret its bits as if it

were an unsigned integer. This has the benefit of not incurring any memory penalties compared to allocating a signed integer, and there is negligible performance difference when performing operations on unsigned integers.

So why does Kotlin not support unsigned floating point numbers? Unsigned numbers are reinterpretations of their signed variants. Although it would be possible to reinterpret a floating point value and change the meaning of the sign bit, this is extremely unconventional – and Kotlin does not support it.

Because of the complexity in performing floating point arithmetic, computers have specialized components that allow them to perform these operations efficiently. If you were to reassign the meaning of the sign bit, you would not be able to use these hardware accelerators, and your application's performance would suffer. To avoid this problem, virtually no languages with unsigned numbers support unsigned floating point values.

Because Kotlin's unsigned types are implemented with the signed types, the signed types tend to be treated as slightly higher-class citizens. You will find that signed types are used overwhelmingly compared to unsigned types. You will also find that they come with their own set of quirks (for example, IntelliJ sometimes treats **UInts** as **Ints** when printing results in the REPL).

Whether and where to use unsigned types is up to you. We have opted not to use them in bounty-board, but you could argue that some variables, like `playerLevel`, should be unsigned, because they should never be negative. Try them out for yourself and see whether you think they would be a good fit for your code.

For the More Curious: Bit Manipulation

Earlier, you saw that numbers have a binary representation. You can get the binary representation for a number at any time. For example, you could ask for the binary representation of the integer 42 with:

```
42.toString(radix = 2)
"101010"
```

(The `radix` parameter indicates the desired base of the outputted number. By specifying 2, you request a number in base 2, or binary. The default radix is 10, which outputs a decimal number. Another commonly used radix is 16, which returns a hexadecimal string.)

Kotlin includes functions for performing operations on the binary representation of a value, called bitwise operations – including operations you may be familiar with from other languages, such as Java, C, and JavaScript. [Table 5.3](#) shows commonly used binary operations in Kotlin.

Table 5.3 Binary operations

Function	Description	Example
shl(bitcount)	Shifts bits left by bitcount.	42.shl(2) 10101000
shr(bitcount)	Shifts bits right by bitcount.	42.shr(2) 1010
inv()	Inverts bits.	42.inv() 1111111111111111111111111111111110101
xor(number)	Compares two binary representations and performs a logical ‘exclusive or’ operation on the corresponding bit positions, returning 1 for each bit position that has a 1 in one input but not the other.	42.xor(33) 001011
and(number)	Compares two binary representations and performs a logical ‘and’ operation on the corresponding bit positions, returning 1 for each bit position that has a 1 in both inputs.	42.and(10) 1010

6

Strings

In programming, textual data is represented by [*strings*](#) – ordered sequences of characters. You have already used Kotlin’s strings for the messages you print when bounty-board is run, like this one:

```
"The hero announces her presence to the world."
```

In this chapter, you will see more of what strings can do. In the process, you will upgrade bounty-board with better string formatting. You will also update the program so the player’s level is read from user input through the console, instead of being hardcoded into the program.

String Interpolation

In the bounty-board project, the hero has a name. It would be better to use the name instead of always referring to “the hero.” You have the constant `HERO_NAME`; you just need to include its value in your strings. One way to accomplish this is [string concatenation](#).

Update the welcome message in **main** to see how this works:

Listing 6.1 Using string concatenation (Main.kt)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5

fun main() {
println("The hero announces her presence to the world.")
    println(HERO_NAME + " announces her presence to the world.")
    println(playerLevel)
    ...
}
...
```

The `+` operator can be used to *concatenate*, or join, two string values. Here, you concatenate the string value of `HERO_NAME` and the string literal `" announces her presence to the world."`. Run bounty-board to see the result:

```
Madrigal announces her presence to the world.
5
...
```

Success! However, Kotlin has another trick up its sleeve that lets you accomplish the same result more concisely. Update your welcome message as shown in [Listing 6.2](#).

Listing 6.2 Using string interpolation (Main.kt)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5

fun main() {
println(HERO_NAME + " announces her presence to the world.")
    println("$HERO_NAME announces her presence to the world.")
    println(playerLevel)
    ...
}
...
```

You have just used a language feature called [string interpolation](#). The `$` symbol introduces a placeholder. [String templates](#) are strings that have one or more placeholders. At runtime, Kotlin will replace the placeholders with values.

When the `$` is immediately followed by a variable name (in this case, `HERO_NAME`), you are telling Kotlin that you want to *interpolate* – insert – the value of the variable into the string. Run your code again to see that the result has not changed.

Using string templates often leads to code that is more readable and concise compared to using string concatenation. To see what we mean, compare these two print statements, which result in the same message being printed to the console:

```
println(HERO_NAME + " is at level " + playerLevel + ".")
println("$HERO_NAME is at level $playerLevel.")
```

Do you find it easier to read the second one? We do, too, which is why we generally prefer this syntax. Spend a moment updating the rest of bounty-board to take advantage of string interpolation.

Listing 6.3 Using more string templates (Main.kt)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5

fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println(playerLevel)

    readBountyBoard()

    println("Time passes...")
println("The hero returns from her quest.")
    println("$HERO_NAME returns from her quest.")

    playerLevel += 1
    println(playerLevel)
    readBountyBoard()
}

private fun readBountyBoard() {
println("The hero approaches the bounty board. It reads:")
    println("$HERO_NAME approaches the bounty board. It reads:")
    println(obtainQuest(playerLevel))
}
...
```

Run your program to make sure the output is what you expect.

You can also use string templates with more complicated expressions. Suppose you wanted to surround the quest name with quotation marks and indent it for emphasis. To accomplish this, update the **println** call as shown in [Listing 6.4](#). We will explain the syntax after you enter it.

Listing 6.4 Using a complex string template (Main.kt)

```
...
private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(observeOn(playerLevel))
    println("\t\t${observeOn(playerLevel)}\n")
}
...
```

There are two new features that you are taking advantage of here. Let’s focus on the string interpolation first. To interpolate the value returned by the **observeOn** function in the string template, you use `${observeOn(playerLevel)}`. Here, the `$` that introduces a string template is followed by curly braces, `{}`.

The full syntax for a string template is `${your-expression-here}`. Inside the curly braces, Kotlin lets you write any expression you want. Line breaks, comments, `if/else` expressions, and `when` expressions are all allowed inside the curly braces.

So why did you not need curly braces for `$HERO_NAME`? If you are only reading a variable, Kotlin lets you omit the braces. For any other expression, you must include them.

Your new string also contains quite a few backslash (`\`) characters. What are they doing there?

As you have seen, the quotation mark character (`"`) signals the start and end of a string literal: `"Madrigal"`. If you typed a string like `"Madrigal proclaims, "Hello, world!"` in your code, the compiler would read the second quotation mark and terminate the string there: `"Madrigal proclaims, "`.

You need to let the compiler know that you want the quotation mark character to be part of the string, not signal its end. To do this, you [escape](#) the quotation mark character, which tells the compiler to treat it as text and not syntax. The backslash is the escape character that accomplishes this.

You are also taking advantage of another [escape sequence](#) with `\t`. The `\t` escape sequence tells Kotlin to insert a tab character. Kotlin has many escape sequences that have special meanings like this. [Table 6.1](#) lists the escape sequences (consisting of `\` and the character being escaped) and their meanings to the compiler.

Table 6.1 Escape sequences

Escape sequence	Meaning
<code>\t</code>	Tab character
<code>\b</code>	Backspace character
<code>\n</code>	Newline character
<code>\r</code>	Carriage return
<code>\"</code>	Double quotation mark
<code>\'</code>	Single quotation mark/apostrophe
<code>\\</code>	Backslash
<code>\\$</code>	Dollar sign
<code>\u</code>	Unicode character

Run bounty-board to see the result of your new string template:

```
Madrigal announces her presence to the world.
5
Madrigal approaches the bounty board. It reads:
"Convince the barbarians to call off their invasion."
Time passes...
Madrigal returns from her quest.
6
Madrigal approaches the bounty board. It reads:
"Locate the enchanted sword."
```

Raw Strings

When you have a string that requires numerous symbols to be escaped, it can be confusing to make sense of what the string is doing. No matter how much you know about each part of the string

"\t\"\${obtainQuest(playerLevel)}\"", it might take you a moment to parse it. Luckily, Kotlin has a more suitable tool for this job: [raw strings](#). A raw string lets you use special characters without escaping them.

Raw strings let you insert newline characters, quotation marks, and other symbols that you would normally need to escape. In fact, raw strings do not support escape sequences at all. You do still have access to string templates when using a raw string, as you will see momentarily. To create a raw string, begin a string with three quotation marks (""") and end the string using another three quotation marks.

To see a raw string in action, update your **readBountyBoard** function to make only one `println` call.

Listing 6.5 Using a raw string (Main.kt)

```
...
private fun readBountyBoard() {
    println("$HERO_NAME approaches the bounty board. It reads:")
    println(obtainQuest(playerLevel))
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        | "${obtainQuest(playerLevel)}"
        | """.trimMargin()
    )
}
...
```

(There are four spaces between the second pipe character [|] and the quotation mark ["], but having a different number of spaces will not cause any problems moving forward.)

The beauty of raw strings is that they preserve whitespace. All your line breaks and any whitespace at the beginning and end of a line will be retained when your code is compiled.

The three quotation marks (""") that introduce and close a raw string do not need to appear on their own line. But for multiline raw strings, many programmers prefer to put the quotation marks on their own line, as you have done here, for readability. Shorter raw strings often appear on a single line, such as """"Welcome, Madrigal!" the mayor proclaimed.""".

The pipe character (|) defines the start of each line of the string. The **trimMargin** function that you called on the raw string looks for lines that start with the pipe and deletes all whitespace to the left of it. The pipe itself is also removed from the final string.

If you omit the pipe and the call to **trimMargin** in a raw string, your text will be indented exactly as you type it in your code. You can make this work by removing all the indentation from your raw strings, but this can harm readability.

Kotlin also has a function called **trimIndent** that works similarly to the pipe character, as long as you want at least one line of your string to be flush left. When called, it will find an amount of leading whitespace that is common to every line and strip it from the string. So you can express the same raw string using **trimIndent** like this:

```
private fun readBountyBoard() {
    println(
        """
        $HERO_NAME approaches the bounty board. It reads:
        "${obtainQuest(playerLevel)}"
        """.trimIndent()
    )
}
```

We have opted for the more explicit route with |s and **trimMargin**, but it is up to you to pick which variation you prefer.

Run bounty-board again. The output will look the same as it did before, but if you inspect the whitespace you will find that the tab character has been replaced by four spaces.

Raw strings can be a major boon for readability compared to trying to parse a string with many escaped characters. It is much more obvious at a glance what this string will look like.

Reading Console Input

Your strings are formatted and include the hero’s name, so it is time to move on to your next task. So far, Madrigal’s skill level has been hardcoded into your application. It can only be changed by modifying your code and recompiling the program. This means that players will always be given the same quest unless they recompile their own game. Not ideal.

To address this shortcoming, you will need to read user input from the console. Update your **main** function with a call to **readLine**, which will do just that.

Listing 6.6 Reading text from the console (Main.kt)

```
...
fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println("What level is $HERO_NAME?")
    val input = readLine()
    println("$HERO_NAME's level is $input.")
    println(playerLevel)

    readBountyBoard()
    ...
}
...
```

Run bounty-board. You will see that your program pauses after printing `What level is Madrigal?`. In IntelliJ’s console window, enter the number 1, then press Return. Your program will then print `Madrigal's level is 1.` and continue executing from there. The full output should be:

```
Madrigal announces her presence to the world.
What level is Madrigal?
1
Madrigal's level is 1.
Madrigal approaches the bounty board. It reads:
    "Convince the barbarians to call off their invasion."
Time passes...
Madrigal returns from her quest.
6
Madrigal approaches the bounty board. It reads:
    "Locate the enchanted sword."
```

You might notice right away that there are some issues with the logic in your code. You will fix those next.

readLine is effectively the opposite of **println**. It returns a **String** with a single line of text that was entered into the console. You can use this function as a way of getting user input for command-line-driven applications.

Converting Strings to Numbers

You are reading in the player’s level, but you can see in the output that you are not yet using it to determine what quests will be given to the player. At level 1, Madrigal should be sent to meet Mr. Bubbles, not to negotiate with the barbarians.

To make better use of the user’s input, you need to assign it to the `playerLevel` var. But before you make any changes to your code, take a moment to inspect the code you just entered.

Move your text cursor over the `input` declaration and press Control-Shift-P (Ctrl-Shift-P) to see what type Kotlin has inferred your new variable to be. The pop-up that appears will tell you that it is a **`String?`**. We will explain what the question mark means in [Chapter 7](#). For now, consider this: `input` is a string, but `playerLevel` is an integer. How do you convert a string into a number so the result of **`readLine`** can be assigned to `playerLevel`?

Luckily, Kotlin’s **`String`** type has a function called **`toInt`** that can perform this conversion automatically. Update the code in `bounty-board` to make this conversion and appropriately set the player’s level.

Listing 6.7 Converting Strings to Ints (`Main.kt`)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5
var playerLevel = 0

fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println("What level is $HERO_NAME?")
    val input = readLine()
    println("$HERO_NAME's level is $input.")
    playerLevel = readLine()!!.toInt()
    println("$HERO_NAME's level is $playerLevel.")

    readBountyBoard()
    ...
}
```

When you call **`toInt`**, the entire string will be parsed and treated as a number. (Wondering about the two exclamation points? This has to do with Kotlin’s null safety techniques, which you will learn more about in [Chapter 7](#).)

Run `bounty-board` again. When prompted for a player level, enter 1. You will see this output:

```
Madrigal announces her presence to the world.
What level is Madrigal?
1
Madrigal's level is 1.
Madrigal approaches the bounty board. It reads:
"Meet Mr. Bubbles in the land of soft things."
Time passes...
Madrigal returns from her quest
2
Madrigal approaches the bounty board. It reads:
"Convince the barbarians to call off their invasion."
```

At this point, everything in the bounty board is now in order. You are asking the player to enter their skill level and printing an appropriate pair of quests. The rest of your work in `bounty-board` will be to refine your code, primarily by adding input validation.

Run `bounty-board` again, but this time enter “one” when asked for Madrigal’s level. Your program will crash and you will see output that looks like this:

```
Madrigal announces her presence to the world.
What level is Madrigal?
one
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "one"
    at java.lang.NumberFormatException.forInputString
        (NumberFormatException.java:68)
    at java.lang.Integer.parseInt(Integer.java:652)
    at java.lang.Integer.parseInt(Integer.java:770)
    at MainKt.main(Main.kt:7)
    at MainKt.main(Main.kt)
```

Run `bounty-board` once more, this time entering 1.0 as Madrigal’s level. Your program will crash again, with a similar error message:

```
Madrigal announces her presence to the world.
What level is Madrigal?
1.0
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "1.0"
    at java.lang.NumberFormatException.forInputString
        (NumberFormatException.java:68)
    at java.lang.Integer.parseInt(Integer.java:652)
    at java.lang.Integer.parseInt(Integer.java:770)
    at MainKt.main(Main.kt:7)
    at MainKt.main(Main.kt)
```

In both of these cases, your program ran into an issue: It was unable to convert the input string into an **`Int`**. You will learn more about how this error is propagated in [Chapter 7](#). For now, think about why Kotlin is unable to convert these inputs.

In the case of “one,” the **`toInt`** function is unable to convert the value because it is not numeric. If you need to parse the spelled-out form of a number, you will need to implement that yourself or use another library – both of which are outside the scope of this book.

To understand why 1.0 cannot be converted to an **Int**, recall what you learned in [Chapter 5](#): Integers are whole numbers that do not have decimal points. So just as you cannot assign 1.0 to an **Int**, you cannot parse the string "1.0" as an **Int**. Instead, you would need to use **toFloat** or **toDouble**, which understand what to do with a decimal point. (Similar functions like **toLong** and **toBigDecimal** exist if you need to parse a very large number or require that your numbers be very precise.)

Regular Expressions

In general, it is a good idea to check inputs before performing “dangerous” operations that could cause your program to crash. There are several approaches you can take to prevent your application from crashing when it receives invalid input. There are also ways to handle errors after they happen, which you will see in [Chapter 7](#).

For now, you will prevent this crash by making sure the user has entered a number before attempting to call **toInt** on it. Specifically, you will ensure that the string returned by **readLine** only contains digits. Kotlin has many tools in place that can help you check that a user’s input is valid. One technique makes use of a [regular expression](#) – a commonly used tool for parsing and working with strings.

A regular expression (also called a *regex*) is a sequence of characters that represents a pattern. Text can be compared against the pattern to find matches. For bounty-board, you can use a regex to see if the input matches a pattern of only digits, as shown in [Listing 6.8](#).

Listing 6.8 Using a regular expression (Main.kt)

```
...
fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println("What level is $HERO_NAME?")
    playerLevel = readLine().toInt()
    val playerLevelInput = readLine()!!
    playerLevel = if (playerLevelInput.matches("""\d+""").toRegex()) {
        playerLevelInput.toInt()
    } else {
        1
    }
    println("$HERO_NAME's level is $playerLevel.")

    readBountyBoard()
    ...
}
...
```

You are using two new functions here: **matches** and **toRegex**. **toRegex** converts a string to an instance of the **Regex** type, which is required by the **matches** function. The string you are passing to **toRegex** is the regular expression `\d+`, which means “one or more digits.” In regex, `\d` means any digit between 0-9, and `+` represents one or more instances of the character to its left.

You are using a raw string, surrounded by `"""`, to define this regular expression. This is optional, but raw strings make it easier to declare regular expressions that use reserved characters like `\`, `$`, `"`, and so on. Recall that raw strings do not support escape sequences. This means that a `\` in a raw string is treated as the backslash character itself, not the beginning of an escape sequence.

If you used a regular string here, you would need to escape the `\` so it would be passed to the **toRegex** function: `"\\d+"`. This would work, but it would make the syntax of the regular expression less clear.

The **matches** function checks that the entirety of the string matches a given regular expression. (Regex can also be used to match sequences within a string, but that is not what you want here.) When you put everything together, the condition `if (playerLevelInput.matches("""\d+""").toRegex())` checks whether all the characters in `playerLevelInput` are digits.

Run bounty-board several times to confirm this. Enter Madrigal’s level as 5, then as 8 to confirm that **toInt** gets called correctly for valid inputs. Then try entering invalid inputs, like “seven,” 3.0, and -5. Your program should never crash, and it should default to level 1 if an invalid input is given.

Regular expressions are a powerful tool when working with strings.

The **Regex** type is available on all platforms supported by Kotlin. You can find more information about the supported regex patterns at docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html. For Kotlin/JS users, see developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Cheatsheet.

Behind the scenes, Kotlin’s **Regex** type uses Java’s **Pattern** class when targeting the JVM. When using Kotlin/JS, the **Regex** type uses the built-in **RegExp** type. For Kotlin/Native users, Kotlin ships with its own internal pattern matching implementation. This native implementation is very similar to Java’s **Pattern** class, so see Java’s **Pattern** documentation for more information on the pattern specifications.

String Manipulation

It turns out that it is taboo to say “Nogartse” aloud (legends claim that saying his name can summon him). Luckily, the **String** type has many functions that allow you to edit stored text while your program is running. Some of these functions also use regular expressions to specify which part of the string should be modified.

In particular, strings have a **replace** function that you can use to censor this taboo name and avoid tempting fate. Add it to **readBountyBoard**’s implementation:

Listing 6.9 Calling replace (Main.kt)

```
private fun readBountyBoard() {
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |  "${obtainQuest(playerLevel).replace("Nogartse", "xxxxxxx")}"
        |""".trimMargin()
    )
}
```

replace takes in two arguments. The first argument selects which parts of a string should be replaced. The second indicates what value the selected portions should be replaced with. **replace** will then find all matches for the selection criteria and overwrite them with the replacement string.

The first argument you are providing to **replace** is "Nogartse". The **replace** function will look for all exact matches of the character sequence “Nogartse” for replacement. You can also provide a **Regex** as a parameter, if you want to replace a pattern that cannot be defined in a simple string. For example, you could check for both the capitalized and lowercase versions of Nogartse with the parameter "[Nn]ogartse".toRegex().

The second argument in your call to **replace** is "xxxxxxx". This will censor Nogartse’s name with an equivalent number of x’s. There are no restrictions on the content of the replacement string. If you wanted to completely scrub Nogartse’s name from all records, you could delete it entirely by passing an empty string ("") for the second parameter.

Putting the two arguments together, your call to **replace** replaces Nogartse’s name with x’s.

Run bounty-board again. This time, when prompted to specify Madrigal’s player level, enter 8. Madrigal will be presented with a censored quest that can safely be spoken.

```
Madrigal announces her presence to the world.
What level is Madrigal?
8
Madrigal's level is 8.
Madrigal approaches the bounty board. It reads:
  "Defeat xxxxxxxx, bringer of death and eater of worlds."
Time passes...
Madrigal returns from her quest.
9
Madrigal approaches the bounty board. It reads:
  "There are no quests right now."
```

Strings are immutable

A clarification regarding the “replacing” of the characters performed by **replace**: Transformation functions like **replace** do not actually modify the original string. They instead return a new string with the modification. To see what we mean, run this code snippet in the REPL:

Listing 6.10 Exploring string immutability (REPL)

```
val finalBoss = "Nogartse"
println(finalBoss) // prints "Nogartse"
val replaced = input.replace("Nogartse", "*****")
println(finalBoss) // prints "Nogartse"
println(replaced)  // prints "*****"
```

Whether they are defined with `var` or `val`, all strings in Kotlin are actually immutable (as they are in Java and JavaScript) – they cannot be changed. Though the variables that hold the value for the **String** can be reassigned if the string is a `var`, the string instance itself can never be changed. Any function that appears to change the value of a string (like **replace**) actually creates a new string with the changes applied to it.

String Comparison

Recall in your **obtainQuest** function that part of your `canTalkToBarbarians` logic checks whether the player is a barbarian.

```
private fun obtainQuest(
    playerLevel: Int,
    hasAngeredBarbarians: Boolean = false,
    hasBefriendedBarbarians: Boolean = true,
    playerClass: String = "paladin"
): String = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    in 2..5 -> {
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")
        ...
    }
    ...
}
```

In this code, you check the [structural equality](#) of `playerClass` and `"barbarian"` using the structural equality operator, `==`. When used with strings, this operation checks whether both strings contain the same text. (You have also seen this operator used with numeric values, where it checks for the same value.)

There is another way to check the equality of two variables: comparing [referential equality](#), which means checking whether two variables share the same reference to a type instance – in other words, that two variables point to the same object in memory. Referential equality is checked using `===`, the [referential equality operator](#).

Referential comparison is not usually what you want. You generally do not care whether strings are the same instance or different instances – only that they have the same characters in the same sequence (that is, their *structures* are identical). Referential equality can also depend on the implementation of the runtime, and may not always behave as desired.

If you are familiar with Java or a C-based language, the string comparison behavior using `==` is likely different than what you expected. Java uses the `==` symbol for referential comparison. A low-level C-based language would also be performing referential comparison in this instance, since this code would normally be comparing pointer values.

When comparing structural equality using the `==` operator, your Kotlin code will be compiled to call a function called **equals** instead. Every type in Kotlin has an **equals** function. This function behaves the same as – and in fact, *is* the same as – Java’s **equals** method. You will learn more about this function in [Chapter 16](#).

In this chapter, you have learned more about how to work with strings in Kotlin. You have seen new ways to declare strings with string templating and raw strings and how to obtain text input from the user. You also saw several techniques that you can use to manipulate strings and transform them into more useful values.

In the next chapter, you will wrap up bounty-board with some final tweaks around input validation and output formatting by leveraging Kotlin’s null safety techniques and exception handling.

For the More Curious: Unicode

As you have learned, a string consists of an ordered sequence of characters. A character is an instance of the **Char** type. Specifically, a **Char** is a [Unicode character](#). The Unicode character encoding system is designed to support “the interchange, processing, and display of the written texts of the diverse languages and technical disciplines of the modern world” (unicode.org).

This means that the individual characters in a string can be any of a diverse palette of characters and symbols – 143,859 of them (and growing) – including characters from the alphabets of every language in the world, icons, glyphs, emoji, and more.

To declare a character, you have two options. Both are wrapped in single quotes. For characters on your keyboard, the simplest option is the character itself in the single quotes:

```
val capitalA: Char = 'A'
```

But not all 143,859 characters are included on your keyboard. The other way to represent a character is with its Unicode character code, preceded by the Unicode character escape sequence `\u`:

```
val unicodeCapitalA: Char = '\u0041'
```

There is a key for the letter “A” on your keyboard, but there is not one for the  symbol. To represent it in your program, your only choice is to use its character code in single quotes. If you want to try it out, open the REPL and execute the code in [Listing 6.11](#).

Listing 6.11 Om... (REPL)

```
val omSymbol = '\u0950'  
print(omSymbol)
```

You will see the  symbol printed in the console.

Null Safety and Exceptions

Some elements in Kotlin can be assigned a value of [null](#). Null is a special value; in fact, it indicates that the value of an element does not exist. In many programming languages, null is a common cause of crashes and errors, because a nonexistent value cannot be asked to do anything. Kotlin has several language features that force programmers to be mindful of null values in their programs, which helps avoid those crashes.

In this chapter, you will learn why null causes a crash, how Kotlin protects against null by default at compile time, and how to safely work with values that could be null when you need them. You will also see how to work with [exceptions](#) in Kotlin: indicators that something went wrong in your program.

To see these issues in action, you will be updating the bounty-board project. You will update your input parsing logic one last time and refine your quest determination logic to better model the scenario where you do not have a quest to assign to the player.

Nullability

As we said, some elements in Kotlin can be assigned a value of null, and some cannot. We say that the former are *nullable* and the latter are *non-nullable*.

Null is not the same as zero or a statement of no value. Think about your bounty board. In the context of a fantasy game, the bounty board likely manifests as a literal bulletin board with quests on it. There is a difference between the board not having a quest (in other words, having a null value) and having a message that says “There are no quests right now.”

While null and “There are no quests right now.” might be interpreted in the same way, they have different implications. A value of null signifies that nothing is present: There are no quests, there is no message – nothing. A value of "There are no quests right now." would mean that this message literally appears on the bounty board.

Likewise, there is also a difference between null and an empty string (""). An empty string might indicate that a blank piece of paper has been posted to the bounty board. But a null value would mean that there is nothing – not even a piece of paper – on the bounty board.

Numeric types in Kotlin can also be nullable, and, similarly, values of null and 0 have different meanings. If you had an **Int** tracking the number of available quests, a value of 0 would mean that there were no available quests. But null would mean that the variable had no value at all – which would mean that the variable’s value could not be incremented or undergo other arithmetic operations until it was reassigned an integer value.

What does all this mean for bounty-board? For the **obtainQuest** function to express that it is possible for the bounty board to not have a quest, you will be updating it to return a nullable string.

Your end result is going to be the same: If there is no quest on the bounty board, you will print a message saying that the bounty board is blank. So why make the change? Nullability does serve a purpose in situations like this. It *forces* you to consider edge cases in your code – like reading from a blank bounty board.

Suppose you wanted to add a feature that reminds Madrigal of her current quest. You might decide to implement it by calling **obtainQuest** again. Without incorporating nullability, you might forget about edge cases like a blank bounty board and accidentally print out "Madrigal’s current quest is: There are no quests right now." But with Kotlin’s nullability features, you are more likely to remember this edge case and instead print "Madrigal does not have a quest now."

Although you will not be implementing a reminder like this, it is still good to get in the habit of asking yourself when your types should be nullable instead of having a value of 0 or an empty string – and it is important to know how to work with nullable types when you need them.

Update your **obtainQuest** function to return a null value in the else branch, as shown in [Listing 7.1](#). (You will see a compiler error after making this change, which we will discuss after you make the update.)

Listing 7.1 Returning a null value (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    ...
    8 -> "Defeat Nogartse, bringer of death and eater of worlds."
    else -> "There are no quests right now."
    else -> null
}
```

Even before you execute this code, IntelliJ warns you with a red underline that something is amiss. Run it anyway, and you will see:

```
Null can not be a value of a non-null type String
```

Kotlin prevents returning a null value from the **obtainQuest** function, because it has a non-null return type (**String**). Non-null types cannot be assigned to null.

This rule does not just apply to functions. This code would fail to compile for the same reason:

```
var quest: String = "Rescue the princess"
quest = null
```

This behavior may be different than what you are familiar with in other languages. In Java, this code is permitted, for example.

```
String quest = "Rescue the princess";
quest = null;
```

Reassigning **quest** to a value of null is fine not only in Java but also in JavaScript and most native languages. But what do you think would happen if you tried to manipulate this string?


```
String quest = "Rescue the princess";  
quest = null;  
quest = quest.replace("princess", "prince");
```

In fact, this code will cause an exception that will crash the program, called a **NullPointerException**. In the world of JavaScript, you are likely familiar with this issue as a **TypeError**. For the native users, you have likely seen the dreaded segmentation fault.

Regardless of platform, this code fails to run because a nonexistent **String** has been asked to replace part of itself. This is an impossible request.

(If you are confused about why a value of null is not the same as an empty string, this example shows the difference: A value of null means that the variable does not exist. An empty string means that the variable exists and has a value of "", which could easily return another empty string as the result of a **replace** call.)

Java and many other programming languages support exactly this pseudocode statement: “Hey, string that may or may not exist, substitute this word with a different one.” In these languages, the value of any variable can be null (with the exception of primitives, which the Kotlin language does not have). In languages that allow null for any type, errors caused by using null values are a common source of application crashes.

Kotlin takes the opposite position on the problem of null. Unless otherwise specified, a variable cannot be assigned a value of null. This guards against the problem of “Hey, nonexistent thing, do something” at compile time, rather than crashing at runtime. (More about compile-time versus runtime errors shortly.)

Kotlin’s Explicit Null Type

NullPointerExceptions like the one you saw above should be avoided at all costs. The Kotlin language and compiler have several techniques that they use to help prevent this kind of crash. First and foremost: Kotlin prevents you from assigning a null value to a variable of a non-nullable type. If you want a value to be nullable, you must explicitly opt in.

To mark the type as nullable, place a question mark after the name of the type, like **String?**. Fix the compiler error introduced in [Listing 7.1](#) by updating the return type of **obtainQuest**:

Listing 7.2 Using a nullable type (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
)String = when (playerLevel) {
): String? = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    ...
    8 -> "Defeat Nogartse, bringer of death and eater of worlds."
    else -> null
}
```

String? is a nullable type. Nullable types are capable of holding either a value of the original type (in this case **String**) or null. You will see how to handle null values in your code momentarily.

The compiler error in **obtainQuest** has now disappeared, and a new one has appeared in its place – this time in **readBountyBoard**. We will revisit this compiler error later in this chapter, but for now, remove the call to **replace** in **readBountyBoard**. (Do not worry, you will reinstate this censorship shortly.)

Listing 7.3 Removing censorship (Main.kt)

```
...
private fun readBountyBoard() {
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |    "${obtainQuest(playerLevel).replace("Nogartse", "xxxxxxx")}"
        |    "${obtainQuest(playerLevel)}"
        |    """.trimMargin()
    )
}
```

Run bounty-board again. When prompted for the player level, enter 10. You will see this output:

```
Madrigal announces her presence to the world.
What level is Madrigal?
10
Madrigal's level is 10.
Madrigal approaches the bounty board. It reads:
>null"
Time passes...
Madrigal returns from her quest.
11
Madrigal approaches the bounty board. It reads:
>null"
```

At level 10, Madrigal has transcended the squabbles of mere mortals, so the bounty-board will not offer her any more quests. To understand what happened, take another look at your **readBountyBoard** implementation.

```
private fun readBountyBoard() {
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |    "${obtainQuest(playerLevel)}"
        |    """.trimMargin()
    )
}
```

Because **obtainQuest** returned a null value when executed, Kotlin inserted the text “null” into the string template. This is better than a crash, but it is not really what you want. You will implement a more polished result later in this chapter. First, you will see how to safely work with nullable values.

To begin, introduce an intermediate variable to store the result of **obtainQuest**. This will give you more control than having the result nested in your raw string.

Listing 7.4 Defining a nullable variable (Main.kt)

```
...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)

    println(
        """
        |$HERO_NAME approaches the bountys board. It reads:
        |    "${obtainQuest(playerLevel)}"
        |    "$quest"
        |    """.trimMargin()
    )
}
```

You declared the type of your new `quest` variable as the nullable **String?** type. If you had used **String** instead, you would get a compiler error when you tried to execute the code: Type mismatch: inferred type is

String? but String was expected. (You can delete the ? to see this for yourself. Be sure to add it back before continuing.)

It is important to understand that **String?** and **String** are two different types. **obtainQuest** now returns a nullable **String?**, not a non-nullable **String**. Kotlin verifies all your assignments to ensure that you do not accidentally assign null values to variables that are not marked as nullable. This happens at compile time to prevent detectable issues before your program even executes.

Compile Time vs Runtime

Kotlin is a [compiled language](#), meaning that your program is translated into machine-language instructions prior to execution by a special program, called the [compiler](#). During this step, the compiler ensures that certain requirements are met by your code before the instructions are generated.

For example, the compiler checks whether null is assigned to a nullable type. As you have seen, if you attempt to assign null to a non-nullable type, Kotlin will refuse to compile your program.

Errors caught at compile time are called [compile-time errors](#), and they are one of the advantages of working with Kotlin. It may sound odd to say that errors are an advantage, but having the compiler check your work during development – before you allow others to run your program and tell you about your mistakes – makes it much easier to track down problems.

On the other hand, a [runtime error](#) is a mistake that happens after the program has compiled and is already running, because the compiler was unable to discover it. For example, because Java lacks any distinction between nullable and non-nullable types, the Java compiler cannot tell you that there is a problem if you ask a variable with a value of null to perform work. Code like that compiles just fine in Java, but it will crash at runtime.

Generally speaking, compile-time errors are preferable to runtime errors. Finding out about a problem while you are writing code is better than finding out later. And finding out after your program has been released? That is the worst.

Null Safety

Because Kotlin distinguishes between nullable and non-nullable types, the compiler is aware of the possibly dangerous situation of asking a variable defined as a nullable type to do something when the variable might not exist. To shield against these dangers, Kotlin will prevent you from calling functions on a value defined as nullable until you have accepted responsibility for this unsafe situation.

To see what this looks like in practice, reinstate your call to **replace** on `quest`:

Listing 7.5 Using a nullable variable (Main.kt)

```
...private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    val censoredQuest = quest.replace("Nogartse", "xxxxxxx")

    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |-----$quest
        |     "$censoredQuest"
        |-----
        """.trimMargin()
    )
}
...
```

The compiler warns you with red underlines about a problem in your new code. (It is the same problem that appeared when you updated the return type of **obtainQuest** to **String?**.) Ignore it and run `Main.kt` anyway. Instead of the censored version of the quest, you will see a compile-time error:

```
Only safe (?.) or non-null asserted (!!) calls
are allowed on a nullable receiver of type String?
```

Kotlin does not allow you to call the **replace** function because you have not dealt with the possibility of `quest` being null. Regardless of whether `quest` has a null or non-null value at runtime, its type remains nullable. Kotlin has prevented you at compile time from potentially causing a runtime error, because the compiler is aware of your mistake with the nullable type.

By now you are likely thinking, “So how do I deal with the possibility of null? I have important business with The Unspeakable One.” You have a number of choices for safely working with a nullable type in Kotlin, and in a moment we will give you three options, plus some extras.

First, though, consider option zero: Ask yourself whether the value in question needs to be nullable.

As we said earlier in this chapter, nullability serves a purpose, and when it is the right choice it is worth the extra work to incorporate nullable types in your code. But if you cannot come up with a good reason to use a nullable type, consider making it non-nullable instead. Non-nullable types are easier to reason about because they are guaranteed to contain a value that can have functions called on it.

Often, it does not make sense for a value to be assigned to a null value. If this is the case, avoiding null is the safest course.

Option one: checking for null values with an if statement

Sometimes, a nullable type is the best tool for a job. In other cases, you will be working with a variable from code you do not control, and you cannot be sure that it will not return null.

The most straightforward option for working safely with null values is a tool you have already learned about: the `if/else` statement. Recall [Table 3.1](#) in [Chapter 3](#), which lists the comparison operators available in Kotlin. The `!=` operator evaluates whether the value on the left is not equal to the value on the right, and you can use it to check that a value is not null. Try it out in **readBountyBoard**:

Listing 7.6 Using `!= null` for null checking (Main.kt)

```
...private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    if (quest != null) {
        val censoredQuest: String = quest.replace("Nogartse", "xxxxxxx")

        println(
            """
            |$HERO_NAME approaches the bounty board. It reads:
            |-----$censoredQuest
            |-----
            """trimMargin()
        )
    }
}
...
```

Now, if `quest` is null, your code will ignore the value and print nothing. Run `bounty-board` and give Madrigal a level of 8 to confirm that it works as expected:

```
Madrigal announces her presence to the world.
What level is Madrigal?
8
```

```

Madrigal's level is 8.
Madrigal approaches the bounty board. It reads:
"Defeat xxxxxxxx, bringer of death and eater of worlds."
Time passes...
Madrigal returns from her quest.
9

```

You may have noticed that IntelliJ highlights `quest` in green when you reference it inside your `if` statement:

```

if (quest != null) {
    val censoredQuest: String = quest.replace("Nogartse", "xxxxxxx")
    ...
}

```

This hint provided by IntelliJ is informing you that you have used a language feature called [smart casting](#). When you check the type of a variable using an `if` statement, Kotlin will automatically [cast](#) it to that type inside the branch – meaning it will treat the variable as though it is the type in question. This same language feature works for null checks: If you verify that a property is not null, Kotlin will automatically cast it to the appropriate non-null type.

However, there are some limitations to smart casting. If you are using a file-level `var`, for example, it is unsafe for Kotlin to perform smart casting because it is possible for the value to change between the time you perform the check and the time you would use the automatically cast value. To get around this, you can make a temporary copy of the variable inside your function scope – or you can use one of the other null safety approaches that Kotlin provides.

Option two: the safe call operator

Although an `if` statement is an intuitive choice for handling null values, it is often not the best option. `if` statements can become awkward and verbose if you only need to call one function on a nullable value or if you have a chain of function calls that can each return a null value.

To clean up this verbosity, you can use the [safe call operator](#) (`?.`) to safely make function calls on nullable objects. Try it out in **readBountyBoard** by moving your `censoredQuest` variable outside the `if` statement:

Listing 7.7 Using the safe call operator (Main.kt)

```

...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    val censoredQuest: String? = quest?.replace("Nogartse", "xxxxxxx")

    if (quest != null) {
        val censoredQuest: String = quest.replace("Nogartse", "xxxxxxx")
    if (censoredQuest != null) {
        println(
            """
            |$HERO_NAME approaches the bounty board. It reads:
            | "$censoredQuest"
            | """.trimMargin()
        )
    }
}
...

```

When the compiler encounters the safe call operator, it knows to check for a null value. At runtime, if the safe call operator is called on a null value, it skips over the call and does not evaluate it, instead returning null.

Here, if `quest` is non-null, a censored version is returned. If `quest` is null, **replace** is not called, because it would not be safe to do so. Run bounty-board and, once again, give Madrigal a level of 8. You should see the censored level 8 quest, and after Madrigal levels up to 9 the program should end without crashing.

The safe call operator ensures that a function is called if and only if the variable it acts on is not null, thus preventing a null pointer exception. We say, using the example above, that **replace** is called “safely,” because the risk of a null pointer exception no longer exists.

Option two and a half: using safe calls with let

Safe calls allow you to call a single function on a nullable type – but what if you want to perform additional work, like creating a new variable or passing the nullable variable as a non-null argument instead of calling a function on it? One way to achieve this is to combine the safe call operator with the function **let**.

let can be called on any value. It creates a new scope with access to the value it is called on, where you can execute code to your heart’s content. You will learn more about **let** in [Chapter 12](#). For now, get a sneak preview by adapting your **readBountyBoard** implementation to use **let** instead of an `if` statement:

Listing 7.8 Using let with the safe call operator (Main.kt)

```

...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    val censoredQuest: String? = quest?.replace("Nogartse", "xxxxxxx")

    if (censoredQuest != null) {
    censoredQuest?.let {
        println(
            """

```



```

        |$HERO_NAME approaches the bounty board. It reads:
        | "$censoredQuest"
        | """.trimMargin()
    }
}
...

```

Right now, you are not getting any advantage by using **let** instead of an if/else statement. In this case, both versions of this code do the same thing (and have the same smart casting limitations we mentioned before). To make better use of the power of **let**, you can consolidate your censorship logic as shown in [Listing 7.9](#).

Listing 7.9 Consolidating code using let (Main.kt)

```

...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
val censoredQuest: String? = quest?.replace("Nogartse", "xxxxxxx")

    censoredQuest?.let {
        println{
    val message: String? = quest?.replace("Nogartse", "xxxxxxx")
    ?.let { censoredQuest ->
        """
        |$HERO_NAME approaches the bounty board. It reads:
        | "$censoredQuest"
        | """.trimMargin()
        }
    }

    println(message)
}
...

```

Here, you define `message` as a nullable variable. You assign its value to the result of safely calling **let** on the `quest` after calling **replace**. When `quest` is not null and **let** is invoked, everything inside the code block following the **let** is executed.

You will learn more about this syntax in [Chapter 8](#) and [Chapter 12](#). For now, understand that inside the **let**’s curly braces, you define a new variable called `censoredQuest` that will have the value from `quest?.replace("Nogartse", "xxxxxxx")`. Because you are using **let** with a safe call operator (`?.`), the value of `censoredQuest` will be non-null, because the safe call operator skips over the **let** call if `quest` is null at runtime.

Run bounty board again, this time specifying that Madrigal is level 6. You should see the same output that you have seen up to this point:

```

Madrigal announces her presence to the world.
What level is Madrigal?
6
Madrigal's level is 6
Madrigal approaches the bounty board. It reads:
"Locate the enchanted sword."
Time passes...
Madrigal returns from her quest.
7
Madrigal approaches the bounty board. It reads:
"Recover the long-lost artifact of creation."

```

The null coalescing operator

Run bounty-board once more, entering 10 for Madrigal’s level. You will see this output:

```

Madrigal announces her presence to the world.
What level is Madrigal?
10
Madrigal's level is 10
null
Time passes...
Madrigal returns from her quest.
11
null

```

A null value is once again making its way into your output, because the safe call to **let** is propagating the null value of `quest` into your `message`. Your next step in bounty-board is to address this issue once and for all. When the bounty-board does not have a quest, it will print out Madrigal approaches the bounty board, but it is blank.

You could accomplish this with an if/else statement that checks whether `message` is null, but Kotlin has another tool that is perfect for this task.

If you want to use a fallback value when a null value is encountered in your code, you can use Kotlin’s [null coalescing operator](#) `?:` (also known as the “Elvis operator” due to its semblance to Elvis Presley’s iconic hairstyle). This operator says, “If the thing on the lefthand side of me is null, do the thing on the righthand side instead.”

Use the null coalescing operator to incorporate your fallback message when the bounty board is empty.

Listing 7.10 Using the null coalescing operator (Main.kt)

```

...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)

val message: String? = quest?.replace("Nogartse", "xxxxxxx")
    val message: String = quest?.replace("Nogartse", "xxxxxxx")
    ?.let { censoredQuest ->
        """
        |$HERO_NAME approaches the bounty board. It reads:
        | "$censoredQuest"
        | """.trimMargin()
    }
}
...

```



```

        } ?: "$HERO_NAME approaches the bounty board, but it is blank."
    println(message)
}
...

```

Most often in this book, we exclude the type of a variable if it can be inferred by the Kotlin compiler. We include `message`’s **String** type here to illuminate the role of the null coalescing operator.

If **let** returns a null value (or if the safe call propagates the null value it was called on), then `message` will be assigned to `"$HERO_NAME approaches the bounty board, but it is blank."`. If `quest` (and, by extension, `censoredQuest`) is not null, then your familiar raw string will be used instead.

Either way, `message` is assigned a value of type **String**, not **String?**. This is great – the bounty board’s message to the user is now guaranteed to be non-null.

Think of the null coalescing operator as ensuring that a value is not null by providing a default non-null value to be assigned if the first option turns out to be null. Null coalescing can be used to clean up values that might be null so that you can have peace of mind as you work with them.

Run `Main.kt` and say that Madrigal is level 10 to see your new output.

```

Madrigal announces her presence to the world.
What level is Madrigal?
10
Madrigal's level is 10
Madrigal approaches the bounty board, but it is blank.
Time passes...
Madrigal returns from her quest.
11
Madrigal approaches the bounty board, but it is blank.

```

The null coalescing operator can also be used independently of the **let** function. You could accomplish the same output with the following code. (Do not make this change to `bounty-board`.)

```

private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)

    val message: String? = quest?.replace("Nogartse", "xxxxxxx")
        ?.let { censoredQuest ->
            """
                |$HERO_NAME approaches the bounty board. It reads:
                |"$censoredQuest"
                |""".trimMargin()
        }

    println(message ?: "$HERO_NAME approaches the bounty board, but it is blank.")
}

```

This example shows an inline usage of the null coalescing operator as an argument to **println**. This code is functionally equivalent to the code in [Listing 7.10](#): If `message` is null, then `"$HERO_NAME approaches the bounty board, but it is blank."` is printed to the console. Otherwise, `message` is printed.

Either way of using the null coalescing operator is valid. So, which style is better? That is not a question that we can answer for you, because the choice is a stylistic one.

There are cases where complex chains of safe calls and null coalescing operators make code harder to read or make the expected result unclear. In these cases, we prefer `if/else` statements for their clarity in sacrifice of conciseness. If you or your team disagree, that is OK – either syntax is valid.

Option three: the non-null assertion operator

The last null-safety technique we will discuss is one that you saw in [Chapter 6](#). It is called the [*non-null assertion operator*](#), and informally it is known as the “double-bang operator” because the symbol uses two exclamation points (`!!`).

The non-null assertion operator (`!!`) can be used to force the compiler to let you call a function on a nullable type. But be forewarned: This is a much more drastic option than the safe call operator and should generally not be used. Visually, the `!!` should look very loud in your code, because it is a dangerous option. If you use `!!`, you are proclaiming to the compiler: “I *demand* you perform this operation! If you cannot do it, then don’t bother executing the rest of my program!”

You are using this operator when you call **readLine**:

```

...
fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println("What level is $HERO_NAME?")
    playerLevel = readLine()!!.replace("[^0-9]".toRegex(), "").toInt()
    println("$HERO_NAME's level is $playerLevel.")
    ...
}
...

```

`readLine()!!.replace(...).toInt()` means, “I don’t care whether **readLine** returns null or not – turn it into a number anyway!” If **readLine** does indeed return null, a **NullPointerException** is thrown.

(This is the case even for Kotlin/JS and Kotlin/Native. Kotlin introduces its own **NullPointerException** type instead of issuing a **TypeError** or causing a segmentation fault. That said, it is still possible to encounter a rogue **TypeError** or segmentation fault in very specific scenarios.)

There are situations where using the double-bang operator is appropriate. Perhaps you do not have control over the type of a variable, but you are sure that it will never be null. As long as you are confident that the variable you are using will not be null when you use it, then `!!` is an option.

In the case of **`readLine`**, a null value will be returned if the user has redirected their input stream to an empty file. If you expect your users to always enter data from the console, this would be an exceedingly rare circumstance. You could very easily justify that your program should not support this use case.

On the flip side, you have other options for your **`readLine`** call, so it might be worth the investment if it means preventing a full application crash. Update your **`main`** function using some of the other null safety techniques you have seen. You can also take advantage of the **`toIntOrNull`** function, which returns null instead of crashing your program if the string cannot be parsed as an integer. Because **`toIntOrNull`** is more graceful about its conversions, there is no longer a need to use a regex to sanitize the input.

Listing 7.11 Removing the double-bang operator (`Main.kt`)

```
...
fun main() {
    println("$HERO_NAME announces her presence to the world")
    println("What level is $HERO_NAME?")
    val playerLevelInput = readLine()!!
    playerLevel = if (playerLevelInput.matches("[0-9]+".toRegex())) {
            playerLevelInput.toInt()
    } else {
            1
    }
    playerLevel = readLine()?.toIntOrNull() ?: 0
    println("$HERO_NAME's level is $playerLevel")
    ...
}
```

Run `bounty-board` a few times, giving it different inputs for Madrigal's level, to test that it works as expected.

Exceptions

Like many other languages, Kotlin includes exceptions to indicate that something went wrong in your program. You have already seen exceptions that get thrown by the system or in the Kotlin standard library. You saw **NumberFormatException** when trying to convert certain strings into an **Int**, and you have seen **NullPointerException** (and the lack thereof) throughout this chapter so far.

But exceptions are not reserved for use by the system. You can use your own exceptions to indicate potential issues in your application.

Throwing an exception

Similar to many other languages, Kotlin allows you to manually signal that an exception has occurred. You do this with the `throw` keyword, and it is called [throwing](#) an exception. There are many more exceptions that can be thrown in addition to the null pointer exception that you just saw.

Why would you want to throw an exception? It is all in the name: Exceptions are used to represent exceptional state. If something in your code has gone extraordinarily wrong, then throwing an exception signals that the issue must be handled before execution continues.

One of the more common exceptions that you will see is the **IllegalArgumentException**.

IllegalArgumentException is a somewhat vague name, to be sure – it means that you provided an input that you have deemed illegal. You can pass **IllegalArgumentException** a string to print out when the exception is thrown to provide more information about what went wrong.

Consider your **obtainQuest** function. You may want to check and ensure that the player's level is not a negative number. It should not be possible to ask for a quest for a level -1 player, because the lowest level that bounty-board is aware of is level 1. You can use an **IllegalArgumentException** to proactively signal when you have used an illegal player level.

You are going to update **obtainQuest** with this check. But first, recall that **obtainQuest** is currently a single-expression function, and you took advantage of some of the reduced syntax Kotlin allows. Because you are adding another statement to this function, you will need to convert its syntax to include a block body.

To do this easily, move your cursor to the function name and press Option-Return (Alt-Enter) to see the intention actions menu. Select Convert to block body, which will automatically insert the curly braces and return keyword. (If the return type were being inferred, IntelliJ would also automatically insert the return type.)

```
...
private fun obtainQuest(
): String? {
    return when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        ...
        else -> null
    }
}
```

Now add a conditional to check that `playerLevel` is not negative:

Listing 7.12 Throwing an **IllegalArgumentException** (**Main.kt**)

```
...
private fun obtainQuest(
): String? {
    if (playerLevel <= 0) {
        throw IllegalArgumentException("The player's level must be at least 1.")
    }

    return when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        ...
        else -> null
    }
}
```

Here, you signal that `playerLevel` must be at least 1; any other input is illegal. This means that anyone that would like to work with the `playerLevel` variable must handle the exceptional state stemming from its minimum allowed value. It is loud, but that is a good thing, as it increases the likelihood that you will notice the exceptional state during development – before it causes a crash for your user.

Run bounty-board and enter -1 for Madrigal's level. You will see the following output:

```
Madrigal announces her presence to the world.
What level is Madrigal?
-1
Madrigal's level is -1.
Exception in thread "main" java.lang.IllegalArgumentException:
    The player's level must be at least 1.
    at MainKt.obtainQuest(Main.kt:41)
    at MainKt.obtainQuest$default(Main.kt:38)
    at MainKt.readBountyBoard(Main.kt:21)
    at MainKt.main(Main.kt:10)
```

```
at MainKt.main(Main.kt)
```

Because you provided an error message to the **IllegalStateException**, you know exactly why your program crashed.

Handling exceptions

Exceptions are disruptive, and they should be – they represent a state that is unrecoverable unless it is handled. Kotlin allows you to specify how to handle exceptions by defining a try/catch statement around the code that might cause one.

The syntax of try/catch is similar to the syntax for if/else. To see what it looks like, use try/catch in **readBountyBoard** to protect against the dangerous operation that you performed:

Listing 7.13 Adding a try/catch statement (Main.kt)

```
...
private fun readBountyBoard() {
    try {
        val quest: String? = obtainQuest(playerLevel)

        val message: String = quest?.replace("Nogartse", "xxxxxxx")
        ?.let { censoredQuest ->
            """
                |$HERO_NAME approaches the bounty board. It reads:
                | "$censoredQuest"
                |""".trimMargin()
            } ?: "$HERO_NAME approaches the bounty board, but it is blank."

        println(message)
    } catch (e: Exception) {
        println("$HERO_NAME can't read what's on the bounty board.")
    }
}
...
```

When you define a try/catch statement, you declare the block of code that you would like to attempt (or “try”) and what should happen if any statement inside the block fails with an exception. If no exception occurs, the try statement executes and the catch statement does not. This branching logic is akin to a conditional.

In the catch block, you define what will happen if some expression in the try block causes an exception. The catch block takes a specific type of exception to protect as an argument. In this case, you catch any exception of type **Exception**, which will catch any error raised by any line of code.

catch blocks can include all sorts of logic, but this example keeps it simple. Here, you use the catch block to print a fallback message describing Madrigal’s newfound literacy predicament.

Now, if you make Madrigal a level -1 character, **obtainQuest** will still throw its **IllegalArgumentException**. But because you handle the exception with a try/catch statement, program execution will continue and the catch block will run, printing the following output to the console. (Try it yourself to confirm.)

```
Madrigal announces her presence to the world.
What level is Madrigal?
-1
Madrigal's level is -1.
Madrigal can't read what's on the bounty board.
Time passes...
Madrigal returns from her quest.
0
Madrigal can't read what's on the bounty board.
```

The message you provide to the **IllegalArgumentException** ("The player's level must be at least 1. ") is no longer printed to the console. The messages you provide when throwing exceptions are intended to be used for debugging purposes, not to be presented to the user. Exception messages often include technical details about your code and are generally unfriendly to most people who will be using your program. You will still see the message if you call **obtainQuest** without using a try/catch block or if you run bounty-board with a debugging tool, such as the one built into IntelliJ.

Try/catch expressions

In [Chapter 3](#), you saw that if/else statements and when blocks could be used as expressions. try/catch blocks can, too. This is an uncommon use case compared to using a conditional expression. Here, it would allow you to clean up redundant **println** calls, using the same technique you first saw in [Chapter 3](#). Go ahead and make these changes so you can see how it works:

Listing 7.14 Using try/catch as an expression (Main.kt)

```
...
private fun readBountyBoard() {
    try {
    val message: String = try {
        val quest: String? = obtainQuest(playerLevel)

val message: String = quest?.replace("Nogartse", "xxxxxxx")
        quest?.replace("Nogartse", "xxxxxxx")
        ?.let { censoredQuest ->
            """
```

```

        |$HERO_NAME approaches the bounty board. It reads:
        |    "$sensoredQuest"
        |    """.trimMargin()
    } ?: "$HERO_NAME approaches the bounty board, but it is blank."

    println(message)
} catch (e: Exception) {
    println("$HERO_NAME can't read what's on the bounty board.")
    "$HERO_NAME can't read what's on the bounty board."
}

println(message)
}
...

```

This change will not affect how your program runs (test it to see for yourself). But it reduces the nearly identical **println** calls into one. It also ensures you continue to handle the exception. If you were to delete the string in the catch block, the compiler would complain that it does not provide a string value to assign to `message`.

This technique can be extremely useful if you want to execute potentially dangerous code to calculate a value and have a fallback option that you can use in its stead.

Preconditions

Unexpected values can cause your program to behave in unintended ways. As a developer, you will spend plenty of time validating input to ensure you are working with the values you intend. Some sources of exceptions are common, like unexpected null values. To make it easier to validate input and debug to avoid certain common issues, Kotlin provides some convenience functions as part of its standard library. They allow you to use a built-in function to throw an exception with a custom message.

These functions are called [precondition functions](#), because they allow you to define preconditions – conditions that must be true before some piece of code is executed.

You have seen a number of ways in this chapter to guard against the **NullPointerException** and other exceptions. One last option is to use a precondition function like **require**, which checks whether a value is null. If the value is not null, **require** returns the value; if it *is* null, the function throws an **IllegalArgumentException**.

Try replacing your thrown **IllegalArgumentException** with a precondition function:

Listing 7.15 Using a precondition function (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String? {
    if (playerLevel <= 0) {
        throw IllegalArgumentException("The player's level must be at least 1.")
    }
    require(playerLevel > 0) {
        "The player's level must be at least 1."
    }

    return when (playerLevel) {
        ...
    }
}
```

Using the **require** function removes the boilerplate of throwing your **IllegalArgumentException**. **require** takes two arguments: The first is a Boolean expression to check. If this Boolean expression evaluates to false, an **IllegalArgumentException** will be thrown. The second argument, which appears in the curly braces, is the error message to be included in the thrown exception. You will learn more about what this set of curly braces means in [Chapter 8](#).

Precondition functions are a great way to communicate requirements before some bit of code is executed. They can be cleaner than manually throwing your own exception, because the condition to be satisfied is included in the name of the function.

Kotlin includes six preconditions in the standard library, shown in [Table 7.1](#):

Table 7.1 Kotlin precondition functions

Function	Description
check	Throws an IllegalStateException if argument is false.
checkNotNull	Throws an IllegalStateException if argument is null. Otherwise returns the non-null value.
require	Throws an IllegalArgumentException if argument is false.
requireNotNull	Throws an IllegalArgumentException if argument is null. Otherwise returns the non-null value.
error	Throws an IllegalArgumentException with a provided message if argument is null. Otherwise returns the non-null value.
assert	Throws an AssertionError if argument is false and the assertion compiler flag is enabled. ^a

^a The details of enabling assertions are outside the scope of this book. If you are interested, see kotlinlang.org/api/latest/jvm/stdlib/kotlin/assert.html and docs.oracle.com/cd/E19683-01/806-7930/assert-4/index.html.

Pay special attention to the **checkNotNull** and **requireNotNull** functions. These two precondition functions take in an argument and throw an exception if it is null. The difference between them is the type of exception they throw: **require** and **requireNotNull** throw **IllegalArgumentException**s, while **check** and **checkNotNull** throw **IllegalStateException**s. Prefer **IllegalArgumentException** when you are checking an input to a function and **IllegalStateException** for most other scenarios.

Precondition functions are another technique to handle null safety. Although the end result is similar to the non-null assertion operator (!!), you can be more expressive. Instead of throwing a generic **NullPointerException**, you can provide more information about what was null.

For example, you might decide to use **checkNotNull** in combination with **readLine**:

```
val input: String = checkNotNull(readLine()) {  
    // Throws an IllegalStateException with the message "No input was provided"  
    "No input was provided"  
}
```

Preconditions are a very convenient way of ensuring your application is running normally – and if it is not running normally, failing as soon as possible to help debug the issue. They can help you easily throw the most commonly used exception types, and we hope you will use them proactively to track down potential problems in your applications.

Run bounty-board one last time and ensure that your precondition is not triggered. Your program's behavior should be the same as before.

In this chapter, you have learned how Kotlin handles problems related to nullity. You have seen that you must explicitly define when you support nullability, because values are by default non-nullable. And you learned that you should favor types that do not support null when possible, because they let the compiler help prevent runtime errors.

You have also seen how to work safely with nullable types when you need them – by using the safe call operator or null coalescing operator or by explicitly checking whether the value is null. You also saw the **let** function and how it can be used in conjunction with the safe call operator to evaluate expressions safely on a nullable variable.

Finally, you learned about exceptions, how to deal with them using the try/catch syntax that Kotlin provides, and how to define preconditions to catch exceptional states before they cause a crash.

This concludes your work in bounty-board. In the next chapter, you will create a new project called NyetHack and learn about a new topic called anonymous functions – a crucial building block for many functions in the Kotlin standard library.

For the More Curious: Custom Exceptions

In this chapter, you saw how to use the `throw` keyword to signal that an exception has occurred. The exception you just threw, **`IllegalArgumentException`**, indicates that an illegal input was provided and gives you the opportunity to add more information by passing a string to be printed when the exception is thrown.

To add more detail to your exception, you can create a custom exception for the particular problem. To define a custom exception, you define a new [class](#) that inherits from some other exception. Classes allow you to define the “things” in your program – monsters, weapons, food, tools, and so on. You will learn lots more about classes in [Chapter 13](#), so do not worry about the details of the syntax now.

Define a custom exception called **`InvalidPlayerLevelException`** in `Main.kt`.

Listing 7.16 Defining a custom exception (`Main.kt`)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String? {
    ...
}

class InvalidPlayerLevelException() :
    IllegalArgumentException("Invalid player level (must be at least 1).")
```

`InvalidPlayerLevelException` is a custom exception that acts as an **`IllegalArgumentException`** with a specific message.

You can throw this new, custom exception in the same way that you threw **`IllegalArgumentException`**, using the `throw` keyword. Throwing your custom exception works exactly the same as it does for other exception types.

Listing 7.17 Throwing a custom exception (`Main.kt`)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String? {
    require(playerLevel > 0) {
        "The player's level must be at least 1."
    }
    if (playerLevel <= 0) {
        throw InvalidPlayerLevelException()
    }

    return when (playerLevel) {
        ...
    }
}
```

`InvalidPlayerLevelException` is a custom error intended to be thrown when `playerLevel` is less than 1. Nothing about the code used to define this exception specifies when it is thrown – that is your responsibility.

Custom exceptions are flexible and useful. You can use them not only to print custom messages but also to add functionality to be executed when the exception is thrown. And they reduce duplication, as you can reuse them across your codebase.

For the More Curious: Checked vs Unchecked Exceptions

In Kotlin, all exceptions are [*unchecked*](#). This means that the Kotlin compiler does not force you to wrap all code that could produce an exception in a try/catch statement.

This may come as a surprise to Java users specifically, because Java supports a mixture of both checked and unchecked exception types. With a checked exception, the compiler checks that the exception is guarded against, requiring you add a try/catch to your program or to explicitly mark that your function throws the exception. Unchecked exceptions are not guarded at compilation time: The compiler will let you throw them regardless of whether you have a corresponding try/catch block.

This *sounds* reasonable. But in practice, the idea of checked exceptions does not hold up as well as the inventors thought it would. Often, checked exceptions are caught (because the compiler requires the checked exception to be handled) and then simply ignored, just to allow the program to compile. This is called “swallowing an exception,” and it makes your program very hard to debug because it suppresses the information that anything went wrong in the first place. In most cases, ignoring the problem at compile time leads to more errors at runtime.

Unchecked exceptions have won out in modern languages because experience has shown that checked exceptions lead to more problems than they solve: code duplication, difficult-to-understand error recovery logic, and swallowed exceptions with no record of an error even taking place.

For this reason, Kotlin does not differentiate between checked and unchecked exceptions. You can throw any exceptions without adding a try/catch block or marking that your function throws an exception. It is up to you to introduce the appropriate safeguards to prevent your application from crashing if something goes wrong at runtime.

Part III

Introduction to Functional Programming and Collections

Functional programming is a paradigm that makes heavy use of higher-order functions: functions that can take in and return other functions to modify their own behavior and the behavior of other functions in your code while it is running. Some programming languages are purely functional, meaning the entire application is structured in this style. While Kotlin is not a pure functional programming language, it does provide many functional tools to allow developers to express logic in a functional paradigm.

In Kotlin, we say that functions are “first-class citizens” – that is, they are treated the same as any other type. The Kotlin standard library takes advantage of this to grant developers a highly flexible and concise set of APIs for common tasks.

In the next five chapters, you will explore more advanced techniques for using functions in Kotlin. You will also explore Kotlin’s collection types, which store groups of data. You will then see how functional programming can let you perform complex operations and algorithms on these collections using just a few lines of code. To see these concepts in action, you will start building a new project called NyetHack – a text-based role-playing game – which you will work on through [Chapter 19](#).

Lambda Expressions and the Function Type

In [Chapter 4](#), you saw how to define functions in Kotlin by naming them and how to call them by name. In this chapter, you will see another way to define functions. This new type of function definition will let you treat a function as a value – much like the way you use **String** and **Int**. With this new style of function definition, you will be able to store functions inside variables, pass functions into other functions, and return functions from functions.

To see these concepts in action, you will begin building a project called NyetHack, which you will work on through [Chapter 19](#).

Introducing NyetHack

Why “NyetHack”? We are glad you asked. Perhaps you remember NetHack, a game released in 1987 by The NetHack DevTeam. NetHack was a single-player text-based fantasy game with ASCII graphics; check it out at nethack.org. You will be building elements of a similar text-based game (no awesome ASCII graphics, though – sorry).

JetBrains, the creator of the Kotlin language, has offices in Russia. In fact, Kotlin is named after a Russian island. When you put together a text-based game like NetHack and Kotlin’s Russian origins, you get NyetHack.

Let’s get started. Open IntelliJ and create a new project. If you already have a project open in IntelliJ, you can select `File → New → Project...` to start the New Project wizard.

In the lefthand column, select Kotlin. In the center pane, select Application under the JVM header and choose Gradle Groovy as your build system. Also, ensure that a Project JDK has been selected. (We recommend any version of Java between 1.8 and 15.) These are the same options you used for bounty-board.

For the project name, enter NyetHack. IntelliJ will propose saving the new project in the same location as your last project; you can change the location if you want. When you are ready, click Next.

The second screen in the wizard lets you specify a bit more information about what template you want to use. You will build NyetHack without a template, so if None is not already selected as the template by default, select it from the dropdown. Press Finish, and IntelliJ will create your empty NyetHack project.

Begin by adding a new file to the project. Find and expand the `src` folder in the project tool window. (You may need to click the NyetHack disclosure arrow to see `src`.) In it, expand the `main` folder, which will expose the `kotlin` folder. Right-click the `kotlin` folder, select `New → Kotlin Class/File`, and create a file (not a class) named `NyetHack`. The new file will open in the editor.

The **main** function, as you saw in [Chapter 1](#), defines the entry point for your program. IntelliJ offers a shortcut for writing this function: Type the word “main” in `NyetHack.kt` and press the Tab key. IntelliJ will automatically add the basic elements of the function for you:

```
fun main() {  
}
```

To kick off your role-playing game, begin by asking the user for their name using the **println** and **readLine** functions.

Listing 8.1 NyetHack’s humble beginning (NyetHack.kt)

```
fun main() {  
    println("A hero enters the town of Kronstadt. What is their name?")  
    val heroName = readLine() ?: ""  
}
```

Run `NyetHack.kt`’s **main** function. You should be prompted in the console to enter a player name, and after you enter it the project will terminate.

Anonymous Functions

The functions that you just called – and in fact, all the functions that you have called up to this point – are regular, [named functions](#). They have all been defined using the `fun` keyword, and they all have a name that is part of their signature. However, you can also define a function without using the `fun` keyword. In fact, you can define functions that do not even have a name.

These types of functions are called [anonymous functions](#), because they do not have a name as part of their definition. Anonymous functions interact with the rest of your code a little differently in that they are commonly passed to or returned from other functions. These interactions are made possible by the *function type*, which you will also learn about in this chapter.

Anonymous functions are an essential part of Kotlin. One way they are used is to allow you to easily customize how built-in functions from the Kotlin standard library work to meet your particular needs. An anonymous function lets you describe additional rules for a standard library function so you can customize its behavior.

Let’s look at an example using the **count** function on the **String** type. By default, **count** returns the total number of characters in a string. But **count** also has a variation that accepts a function as a parameter. This function has a single **Char** parameter and returns a **Boolean**. This version of the **count** function calls the provided function on every character in the **String** and returns the total number of true results returned.

To see this in action, enter the following code in the REPL:

Listing 8.2 Counting letters in a string (REPL)

```
"Mississippi".count({ letter -> letter == 's' })
```

Press Command-Return (Ctrl-Enter) to execute this code. You should see 4 printed in the REPL.

Check out the new syntax you are using inside the parentheses of the **count** function:

```
{ letter -> letter == 's' }
```

This piece of syntax is called a [lambda expression](#) (also sometimes called a [function literal](#)), and it is used to create an anonymous function. This lambda expression represents a function that you are passing into the **count** function.

From here on, we will refer to anonymous functions as [lambdas](#) and their definitions as lambda expressions. This is common terminology you will encounter in the wild as well. (A bit of trivia: Why “lambda”? The term, also represented with the Greek character λ , is short for “lambda calculus” – a system of logic for expressing computations, devised in the 1930s by mathematician Alonzo Church. You use lambda calculus notation when you define an anonymous function.)

To understand how **count** works, take a closer look at Kotlin’s lambda expression syntax by defining your own. You are going to write a small helper file to develop a basic narrator. This narrator will have different moods, which will be implemented using lambda expressions.

Lambda Expressions

In your NyetHack project, create a new file called `Narrator.kt` to host your narrator and its moods. For now, your narrator will only have one mood: loud. Create a new function called **narrate** and implement it by defining a lambda expression, calling it, and printing the result:

Listing 8.3 Defining a lambda expression (Narrator.kt)

```
fun narrate(
    message: String
) {
    println({
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }())
}
```

Just as you write a string by putting characters between opening and closing quotes, you write a function by putting an expression or statements between opening and closing curly braces. Here, you begin with a call to **println**. Inside the parentheses that enclose **println**'s argument, you define an anonymous function inside a set of curly braces using the lambda expression syntax. The lambda expression defines a variable and returns a transformed version of the `message` argument:

```
{
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}
```

Outside the lambda expression's closing brace, you call the function with a pair of empty parentheses. If you were to leave the parentheses off the end of the lambda expression, the greeting message string would not print. Just like a regular function, an anonymous function does its work only when it has been called, using parentheses along with any arguments the function expects (zero, in this case):

```
{
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}()
```

To put your new narrator to use, replace your **println** call in `NyetHack.kt` with a call to **narrate**.

Listing 8.4 Calling narrate (NyetHack.kt)

```
fun main() {
    println("A hero enters the town of Kronstadt. What is their name?")
    narrate("A hero enters the town of Kronstadt. What is their name?")
    val heroName = readLine() ?: ""
}
```

Run the **main** function again. You will see the following output:

```
A HERO ENTERS THE TOWN OF KRONSTADT. WHAT IS THEIR NAME?!!!
```

The function type

In [Chapter 2](#), you learned about data types like **Int** and **String**. Function literals also have a type, called the [function type](#). Variables of the function type hold a function as their value, and the function can then be passed around your code like any other variable.

(Do not confuse the function type with a type called **Function**. You define the specifics of a function using a function type declaration, which varies depending on the details of a particular function's input, output, and parameters, as you will soon see.)

Update `Narrator.kt` to define a variable that holds a function, and assign it the lambda expression that formats the message. There is some unfamiliar syntax here, which we will explain after you enter it.

Listing 8.5 Assigning the lambda expression to a variable (Narrator.kt)

```
fun narrate(
    message: String
) {
    val narrationModifier: () -> String = {
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }

    println({
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }())
    println(narrationModifier())
}
```

When you declare a variable, you follow its name with a colon and its type when you need or want to make the type explicit. That is what you have done here with `narrationModifier: () -> String`. Just as `: Int` tells the compiler what kind of data a variable can hold (an integer), the function type declaration `: () -> String` tells the compiler that the `narrationModifier` variable should hold a function.

A function type definition consists of two parts: the function's parameters, in parentheses, followed by its return type, delimited by the arrow (->), as shown in [Figure 8.1](#).

Figure 8.1 Function type syntax

```
fun narrationModifier(): String
    ↓      ↓
    () -> String
```

The type declaration you specified for the `narrationModifier` variable, `() -> String`, indicates to the compiler that `narrationModifier` can be assigned any function that accepts no arguments (indicated by the empty parentheses) and returns a **String**. As with any other type declaration for a variable, the compiler will ensure that the function assigned to the variable or passed as an argument is of the correct type.

Run **main** and confirm that you see the same output:

```
A HERO ENTERS THE TOWN OF KRONSTADT. WHAT IS THEIR NAME?!!!
```

Implicit returns

You may have noticed that there is no return keyword in the lambda expression you defined:

```
val narrationModifier: () -> String = {
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}
```

However, the function type you specified indicates that the function must return a **String**, and the compiler did not complain. And, based on the output, a string is indeed returned: the modified welcome message. Why, then, is there no return keyword?

Unlike regular functions, the lambda expression syntax does not require – or even allow, except in rare cases – the return keyword to output data. Lambda expressions *implicitly* return the last line of their function definition, allowing you to omit the return keyword.

This feature of lambda expressions is both a convenience and a necessity of the syntax. The return keyword is prohibited in a lambda expression because it could be ambiguous to the compiler which function the return is from: the lambda expression itself, or the function that called it.

Function arguments

Like other functions, a lambda can accept zero, one, or multiple arguments of any type. The parameters a lambda accepts are indicated by type in the function type definition and then named in the lambda's definition.

Because your narrator's mood should be persistent, the `narrationModifier` function should be declared as a top-level variable instead of inside the **narrate** function. Make this refactor, updating the `narrationModifier` variable declaration to accept the message as an argument:

Listing 8.6 Adding a message parameter (Narrator.kt)

```
val narrationModifier: (String) -> String = { message ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}

fun narrate(
    message: String
) {
    val narrationModifier: () -> String = {
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
}

    println(narrationModifier(message))
}
```

Here you specify that the lambda accepts a **String** by placing the argument type in the function type's parentheses:

```
val narrationModifier: (String) -> String = { message ->
```

You name the string parameter within the function, right after the opening curly brace:

```
val narrationModifier: (String) -> String = { message ->
```

Lambda expressions that provide parameter names in this way separate them from the function body using the arrow operator (->).

Run `Nyethack.kt` again. The output will not change, but your lambda expression is now taking in the `message` argument itself instead of reading it from **narrate**'s argument.

Remember the **count** function, which can take a function to check against each character in a string? That function is a [predicate](#) argument, called **predicate**, of type `(Char) -> Boolean` – in other words, a function that takes a **Char** argument and returns a **Boolean**. You will see function types and lambdas throughout much of the Kotlin standard library.

The it identifier

When defining lambdas that accept exactly one argument, the `it` identifier is available as a convenient alternative to specifying the parameter name. Both `it` and a named parameter are valid when you have a lambda that has only one parameter.

Delete the parameter name and arrow from the beginning of your `narrationModifier` lambda and use the `it` identifier instead:

Listing 8.7 Using the `it` identifier (`Narrator.kt`)

```
val narrationModifier: (String) -> String = { message ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
    it.uppercase() + "!"
}

fun narrate(
    message: String
) {
    println(narrationModifier(message))
}
```

Run `NyetHack.kt` to confirm that it works as before.

`it` is convenient in that it requires no variable naming, but it is not very descriptive about the data it represents. We suggest that when you are working with more complex lambda expressions, or with nested lambdas (lambdas within lambdas), you stick with naming the parameter to preserve future readers’ (and your own) sanity.

On the other hand, `it` is great for shorter expressions. For example, it would allow the **count** function call you saw earlier, to count the s’s in “Mississippi,” to be written more concisely, like this:

```
"Mississippi".count({ it == 's' })
```

Because of the simplicity of this example, this logic is clear even without an argument name.

Accepting multiple arguments

While the `it` syntax is available for a lambda that accepts one argument, it is not allowed when there is more than one argument. However, lambdas can certainly accept multiple named arguments.

Suppose you wanted to tweak `narrationModifier` so the tone of the message influences how the narrator relates it. Try this out by running the code in [Listing 8.8](#) in the REPL. (For simplicity’s sake, you will not be making this change in `NyetHack`.)

Listing 8.8 Accepting a second argument (REPL)

```
val loudNarration: (String, String) -> String = { message, tone ->
    when (tone) {
        "excited" -> {
            val numExclamationPoints = 3
            message.uppercase() + "!"
        }
        "sneaky" -> {
            "$message. The narrator has just blown Madrigal's cover.".uppercase()
        }
        else -> message.uppercase()
    }
}

println(loudNarration("Madrigal cautiously tip-toes through the hallway", "sneaky"))
```

Press Command-Return (Control-Enter) to run this code. You will see this output in the REPL:

```
MADRIGAL CAUTIOUSLY TIP-TOES THROUGH THE HALLWAY. THE NARRATOR HAS JUST BLOWN
MADRIGAL'S COVER.
```

This lambda expression declares two parameters, `message` and `tone`, and accepts two arguments when called. Because there is more than one parameter defined for the expression, the `it` identifier is no longer available.

Unlike regular functions, lambdas cannot have default arguments. The function type is the only information Kotlin retains for this kind of function, and it is not possible to include a default argument in a function’s type as you can with a regular function. Named arguments are similarly disallowed when using a lambda.

Type Inference Support

Kotlin’s type inference rules apply to function types much like they do with the types you met earlier in this book: If a variable is given a lambda as its value when it is declared, no explicit type definition is needed.

This means that the lambda you wrote earlier that accepted no arguments:

```
val narrationModifier: () -> String = {
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}
```

could also have been written with no specified type, like this:

```
val narrationModifier = {
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}
```

Type inference is also an option when the lambda accepts one or more arguments, with a slight catch. The compiler needs help to figure out what types the parameters of your lambda expression are. When using type inference, you must provide both the name and the type of each parameter in the lambda expression’s definition. This also means that it is not an option to use the `it` shorthand if you want to use type inference.

Update the `narrationModifier` variable to use type inference by including the parameter type.

Listing 8.9 Using type inference for `narrationModifier` (`Narrator.kt`)

```
val narrationModifier: (String) -> String = {
val narrationModifier = { message: String ->
    val numExclamationPoints = 3
    it.uppercase() + "!".repeat(numExclamationPoints)
    message.uppercase() + "!".repeat(numExclamationPoints)
}

fun narrate(
    message: String
) {
    println(narrationModifier(message))
}
```

Run `NyetHack.kt` and confirm that it works just as before.

When combined with an ambiguous implicit return type, type inference may make a lambda expression difficult to read. But when your lambda expressions are simple and clear, type inference is an asset for making your code more concise.

More Effective Lambdas

Take another look at your `Narrator.kt` file:

```
val narrationModifier = { message: String ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}

fun narrate(
    message: String
) {
    println(narrationModifier(message))
}
```

Eagle-eyed readers may have noticed a disappointing reality: Everything that you have done so far could have been accomplished without using lambdas. This same logic could have been expressed with a single non-anonymous function, like this:

```
fun narrate(
    message: String
) {
    val numExclamationPoints = 3
    println(message.uppercase() + "!".repeat(numExclamationPoints))
}
```

Fear not: The work you have done so far has not been in vain. Lambdas shine when you need to change a function's behavior. Now that you have the basics of lambdas down, you can flesh out `NyetHack`'s narration to include multiple moods.

You have a number of changes to make. You need to make your `narrationModifier` a var, so you can reassign it, and remove the default modification. Then, you will create a new function called **`changeNarratorMood`** to randomly assign a new narrator mood to this variable. And to do that, you will need to import the **`Random`** type and its **`nextInt`** function:

Listing 8.10 Adding more narrator moods (`Narrator.kt`)

```
import kotlin.random.Random
import kotlin.random.nextInt

val narrationModifier = { message: String ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}
var narrationModifier: (String) -> String = { it }

fun narrate(
    message: String
) {
    println(narrationModifier(message))
}

fun changeNarratorMood() {
    val mood: String
    val modifier: (String) -> String

    when (Random.nextInt(1..4)) {
        1 -> {
            mood = "loud"
            modifier = { message ->
                val numExclamationPoints = 3
                message.uppercase() + "!".repeat(numExclamationPoints)
            }
        }
        2 -> {
            mood = "tired"
            modifier = { message ->
                message.lowercase().replace(" ", "... ")
            }
        }
        3 -> {
            mood = "unsure"
            modifier = { message ->
                "$message?"
            }
        }
        else -> {
            mood = "professional"
            modifier = { message ->
                "$message."
            }
        }
    }

    narrationModifier = modifier
    narrate("The narrator begins to feel $mood")
}
```

Now, update your **`main`** function in `NyetHack.kt` to make use of the new moods.

Listing 8.11 Changing the narrator's mood (`NyetHack.kt`)

```
fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?")
    val heroName = readLine() ?: ""

    changeNarratorMood()
    narrate("$heroName heads to the town square")
}
```

Run `NyetHack` a few times. You can enter any name you like for the hero name. `Random.nextInt(1..4)` will return a random number between 1 and 4. Depending on what number is chosen, you will see one of the following outputs.

If the random number generator returns 1:

```
A hero enters the town of Kronstadt. What is their name?
Madrigal
THE NARRATOR BEGINS TO FEEL LOUD!!!
```

MADRIGAL HEADS TO THE TOWN SQUARE!!!

If the random number generator returns 2:

```
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
the... narrator... begins... to... feel... tired  
madrigal... heads... to... the... town... square
```

If the random number generator returns 3:

```
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
The narrator begins to feel unsure?  
Madrigal heads to the town square?
```

If the random number generator returns 4:

```
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
The narrator begins to feel professional.  
Madrigal heads to the town square.
```

narrate is now leveraging lambdas in their fullest capacity. You can arbitrarily change the narrator’s behavior by assigning **narrationModifier** as desired. This has one incredible benefit that you cannot achieve without lambdas: You can change how **narrate** works without changing its implementation. This is an extremely powerful technique that you will see throughout the Kotlin standard library, and likely a pattern that you will adopt in your own code.

Defining a Function That Accepts a Function

You have already seen a few ways that lambdas can customize the work of standard library functions. The **count** function, for example, can take in a lambda to influence what it counts. You can also pass lambdas into functions that you write yourself.

A function parameter can accept arguments of any type, including arguments that are functions. A function-type parameter is defined like a parameter of any other type: You list it in the parentheses after the function name and include the type. To see how this works, you will add a new argument to **narrate** that will override the narrator’s current mood to allow for one-off formatting.

Add a parameter called **modifier** to the **narrate** function in `Narrator.kt` with the type `(String) -> String`. Because this parameter will only be used for some specific narration calls, give it a default value to use the `narrationModifier` you have been working with so far.

Listing 8.12 Adding a function argument to narrate (Narrator.kt)

```
import kotlin.random.Random
import kotlin.random.nextInt

var narrationModifier: (String) -> String = { it }

fun narrate(
    message: String
    message: String,
    modifier: (String) -> String = { narrationModifier(it) }
) {
    println(narrationModifier(message))
    println(modifier(message))
}
...
```

The **modifier** parameter you added to **narrate** is a function that accepts a **String** and returns a **String**. It serves the same role as `narrationModifier`, but it can be overridden for a specific call to **narrate**.

To exercise your new lambda expression, you will introduce color into the world of NyetHack by making the welcome message (“A hero enters the town of Kronstadt. What is their name?”) print in yellow. Many terminals – including the one built into IntelliJ – support basic text styling using ANSI escape sequences. ANSI escape sequences date back to the 1970s; they signal information an ANSI-compatible terminal can use to provide a (slightly) richer output to users.

There are two ANSI escape sequences you will use in your code. These sequences may look intimidating at first, but stick with us – we will explain what each part does. Update your **main** function in `NyetHack.kt` to print the first message to the user in yellow.

Listing 8.13 Making the welcome text yellow (NyetHack.kt)

```
fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?", { message ->
        // Prints the message in yellow
        "\u001b[33;1m$message\u001b[0m"
    })
    val heroName = readLine() ?: ""

    changeNarratorMood()
    narrate("$heroName heads to the town square")
}
```

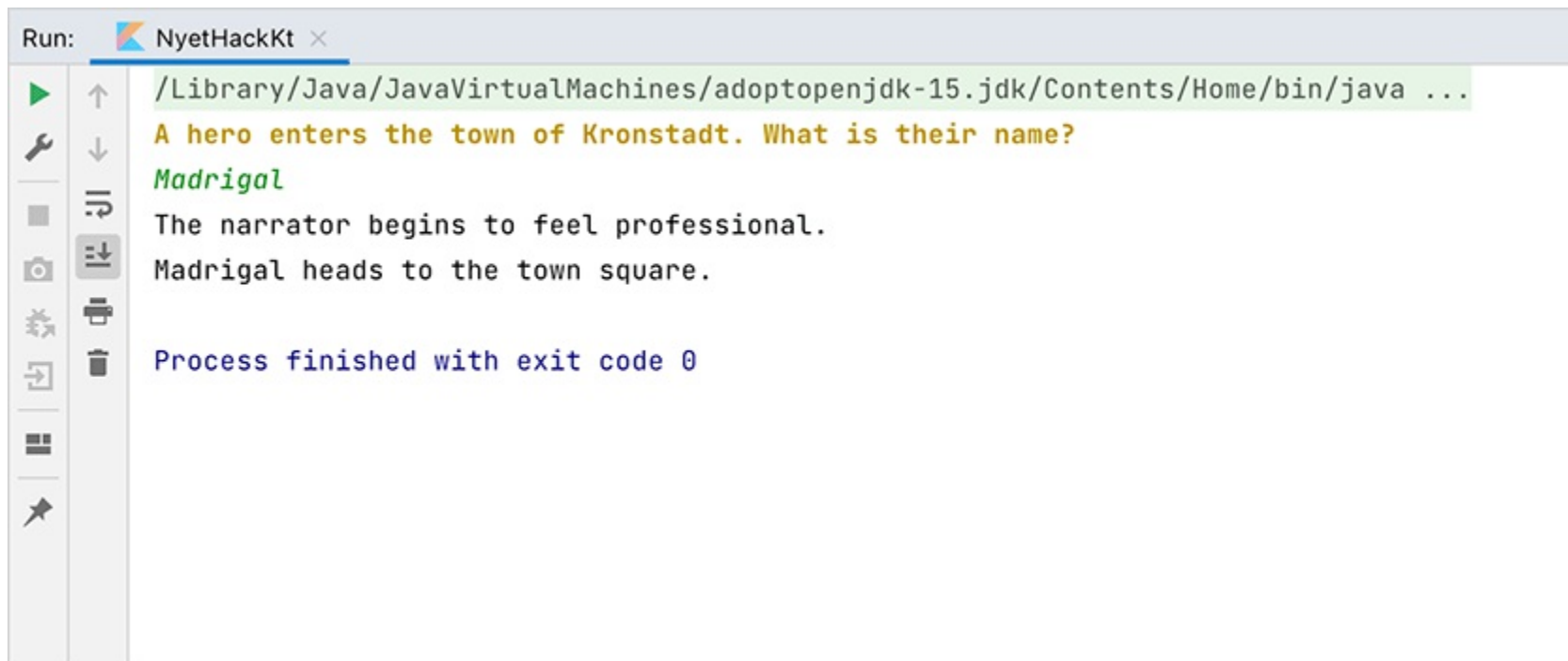
The first ANSI escape sequence, `\u001b[33;1m`, indicates that you want all the text that follows it to be yellow. `\u001b` uses the Unicode escape syntax you saw in [Chapter 6](#) to signal an ANSI escape sequence. The bracket (`[]`) represents the start of the command, and `33;1` corresponds to the text color “bright yellow,” as defined by the terminal’s color scheme. The `m` is the last part of the command and indicates that you are changing text modes. Put it all together and you get, “Change the text style from this point forward so the color is bright yellow.”

Because you do not want *all* your text to appear in yellow, you reset your text style after the welcome message with the ANSI escape sequence `\u001b[0m`. The only difference between the two commands is what comes between the bracket (`[]`) and the `m`. Here, the value of `0` causes the text to reset to its default mode.

Run NyetHack and admire the bright yellow initial text ([Figure 8.2](#) – trust us, it is yellow!).

(If you have set your IDE or terminal to use a color scheme other than the default, the text may print in some other color. This is OK. As long as the opening line prints in a different color than the rest of the text in the console, your code is working as it should.)

Figure 8.2 NyetHack, now in color



Trailing lambda syntax

When a function accepts a function type for its last parameter, you can omit the parentheses around the lambda argument. So this example that we showed you earlier:

```
"Mississippi".count({ it == 's' })
```

can also be written this way, without the parentheses:

```
"Mississippi".count { it == 's' }
```

This syntax is cleaner to read and gets to the essential ingredients of your function call just a bit more quickly.

This simplification can be made only when a lambda is passed as the last argument into a function. When writing functions, declare function type parameters as the final parameter so callers of your function can take advantage of this pattern.

In NyetHack, you can take advantage of this shorthand with your **narrate** function. **narrate** expects two arguments: a string and a function. Refactor your code to provide the arguments that are not functions inside parentheses. Then, list the last argument, the function, outside the parentheses:

Listing 8.14 Passing a trailing lambda (NyetHack.kt)

```
fun main() {  
    narrate("A hero enters the town of Kronstadt. What is their name?", { message ->  
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->  
        // Prints the message in yellow  
        "\u001b[33;1m$message\u001b[0m"  
    }  
    val heroName = readLine() ?: ""  
    ...  
}
```

Nothing changed in the implementation of your **main** function; all that has changed is how it is called. Trailing lambda syntax empowers you to write cleaner code, and we will use it throughout this book.

Function Inlining

Lambdas are useful because they enable a high degree of flexibility in how your programs can be written. However, that flexibility comes at a cost.

When you define a lambda, it is represented as an instance of an object, regardless of which platform your Kotlin code is running on. Also, at runtime, your program performs memory allocations for all variables accessible to the lambda, and this behavior comes with associated memory costs. As a result, lambdas introduce memory overhead that can in turn cause a performance impact – and, in general, performance impacts are to be avoided.

Fortunately, there is an optimization you can enable that removes the overhead when using lambdas as arguments to other functions, called [inlining](#). Inlining removes the need for your lambda to use an object instance and perform variable memory allocations for itself.

To inline a lambda, you mark the function that accepts the lambda using the `inline` keyword. Add the `inline` keyword to the **narrate** function in `Narrator.kt`:

Listing 8.15 Using the `inline` keyword (`Narrator.kt`)

```
...
inline fun narrate(
    message: String,
    modifier: (String) -> String = { narrationModifier(it) }
) {
    println(modifier(message))
}
...
```

Now that you have added the `inline` keyword, instead of invoking **narrate** with a lambda object instance, the compiler “copy and pastes” the function body where the call is made. Take a look at the decompiled Kotlin bytecode for `NyetHack.kt`’s **main** function, where the (now inlined) **narrate** function is called:

```
...
public static final void main() {
    String message = "A hero enters the town of Kronstadt. What is their name?";
    String var2 = "\u001b[33;1m" + message + "\u001b[0m";
    System.out.println(var2);
    String var10000 = ConsoleKt.readLine();
    if (var10000 == null) {
        var10000 = "";
    }

    String heroName = var10000;
    NarratorKt.changeNarratorMood();
    String message$iv = heroName + " heads to the town square";
    String var4 = (String)NarratorKt.getNarrationModifier().invoke(message$iv);
    System.out.println(var4);
}
...
```

Instead of invoking the **narrate** function, **narrate**’s function body is now directly inserted into the **main** function, avoiding the need to pass any lambda at all (and so avoiding the need for a new object instance).

Function inlining is an easy way to get a quick performance boost in your application, and we recommend using it whenever possible for functions that accept lambdas.

This is so easy that you might be asking yourself: Why are functions not inlined by default? There are several scenarios in which it is not possible to declare a function with the `inline` keyword. Here are two common scenarios that can prevent a function from being inlined:

- The function is recursive (meaning that it calls itself). Recursive functions cannot be inlined because the function would need to be inlined into itself, meaning that you would need to insert an infinite number of copies of the original function.
- The function uses functions or variables that have a more restrictive visibility scope than the function being inlined. Your function bodies need to be inserted in their entirety everywhere an inlined function is called. This inserted code is subject to the same visibility rules as normal code, meaning that if you have a public inline function, all the functions and variables it uses must also be public. However, you can make a private inline function that uses private functions and variables.

There are reasons beyond lambdas that you might mark as a function as `inline`. You will see another use case in [Chapter 18](#).

Lambdas and the Kotlin Standard Library

Now that you understand the fundamentals of how to declare and call functions that use lambdas, let’s dive deeper into the Kotlin standard library to see how they appear throughout the language. Update your **main** function with a call to **require**, which you saw for the first time in [Chapter 7](#).

Listing 8.16 Remembering require (NyetHack.kt)

```
fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Prints the message in yellow
        "\u001b[33;1m$message\u001b[0m"
    }
    val heroName = readLine() ?: ""

    val heroName = readLine()
    require(heroName != null && heroName.isNotEmpty()) {
        "The hero must have a name."
    }

    changeNarratorMood()
    narrate("$heroName heads to the town square")
}
```

As you may have already pieced together, the curly braces that appear after the **require** call are a lambda expression. This particular lambda expression is called **lazyMessage**, and its type is `() -> Any`.

require will check the condition you provide as the first argument. If it is false, it will throw an **IllegalArgumentException** with the message generated by your lambda expression. (The lambda **lazyMessage** is optional, by the way. If you do not provide a value, the default is blank.) Here, the lambda expression is used as a performance optimization: The string is only computed when it is needed to throw the exception.

lazyMessage is not unique to **require**. In fact, it is available for almost all the preconditions you saw in [Chapter 7](#), including **requireNotNull**, **check**, **checkNotNull**, and **assert**. (**error** is special: It is the only precondition function that does not have a lazy message. Instead, it takes in its message as a **String**.)

In addition to providing some clever performance optimizations in its standard library, Kotlin also includes many functions that let you express complex algorithms with a single line of code.

Suppose you want to give the player a title that is influenced by their name, like “The Master of Vowels,” for a name with many vowels, or “The Identifiable,” for a name consisting of digits. Kotlin’s **String** type includes several functions that take lambda arguments that come in handy when making determinations like this.

Add a **createTitle** function under **main**. Begin by checking whether the player’s name has more than four vowels, and assign the title “The Master of Vowels” when it does.

Listing 8.17 Defining createTitle (NyetHack.kt)

```
fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Prints the message in yellow
        "\u001b[33;1m$message\u001b[0m"
    }

    val heroName = readLine()
    require(heroName != null && heroName.isNotEmpty()) {
        "The hero must have a name."
    }

    changeNarratorMood()
    narrate("$heroName heads to the town square")
    narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
}

private fun createTitle(name: String): String {
    return when {
        name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowels"
        else -> "The Renowned Hero"
    }
}
```

Run NyetHack and enter a name with many vowels, like Aurelia. Your output will look something like this (depending on what mood the narrator is in):

```
A hero enters the town of Kronstadt. What is their name?
Aurelia
The narrator begins to feel professional.
Aurelia, The Master of Vowels, heads to the town square.
```

Another helpful **String** function is **all**, which checks whether every character in a string matches a given predicate (of type **(Char) -> Boolean**). There is also a **none** function that checks the opposite – that none of the characters in a string match a given predicate.

Using **all** and **none**, implement two more titles for your player. If a player’s name is composed exclusively of numbers, they should be assigned the title “The Identifiable.” If a player’s name does not have any letters, they should be assigned the title “The Witness Protection Member.” Numbers are not letters, of course, so order your conditions within the when expression carefully to make sure that all the titles surface when you intend them to.

Listing 8.18 Adding more titles (NyetHack.kt)

```
...
private fun createTitle(name: String): String {
    return when {
        name.all { it.isDigit() } -> "The Identifiable"
        name.none { it.isLetter() } -> "The Witness Protection Member"
        name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowels"
        else -> "The Renowned Hero"
    }
}
```

Test your code with inputs like “11” and “*****” to ensure that they are given the appropriate title.

all, **none**, and **count** perform checks like these concisely. Without these functions, it would be up to you as a developer to implement these algorithms (likely using 5-10 lines of code per condition).

Kotlin’s standard library includes many functions like these that can make your code more concise and understandable. You will learn more about these functions and get a taste of a paradigm called functional programming in [Chapter 11](#).

For the More Curious: Function References

Up to this point, you have used lambdas to provide a function as an argument to another function. There is another way to do so: by passing a [function reference](#). A function reference converts a regular function defined using the `fun` keyword into a value with a function type. You can use a function reference anywhere you use a lambda expression.

Suppose that you wanted to extract the lambda expression that prints your text in yellow. To make this change, you would extract the lambda expression's implementation into its own function and use the function reference syntax (highlighted), like this:

```
fun main() {
    narrate(
        "A hero enters the town of Kronstadt. What is their name?",
        ::makeYellow
    )
    ...
}

private fun makeYellow(message: String) = "\u001b[33;1m$message\u001b[0m"

private fun createTitle(name: String): String {
    ...
}
```

To obtain a function reference, you use the `::` operator with the function name you would like a reference for. Function references are useful in a number of situations. If you have a named function that fits the needs of a parameter that requires a function argument, a function reference allows you to use it instead of defining a lambda. Or you may want to use a Kotlin standard library function as an argument to a function. Function references make these even more concise than a typical lambda expression.

For the More Curious: Capturing Lambdas

In Kotlin, a lambda can modify and reference variables defined outside its scope. Any variables the lambda accesses that are defined outside it are *captured* by the lambda, meaning that the lambda will hold a reference back to it. This was the case of the first lambda expression you used in your **narrate** function.

As a demonstration of this property of lambdas, add another mood to the **changeNarratorMood** function:

Listing 8.19 Modifying variables from a lambda (Narrator.kt)

```
...
fun changeNarratorMood() {
    val mood: String
    val modifier: (String) -> String

    when (Random.nextInt(1..4)) {
    when (Random.nextInt(1..5)) {
        ...
        3 -> {
            mood = "unsure"
            modifier = { message ->
                "$message?"
            }
        }
        4 -> {
            var narrationsGiven = 0
            mood = "like sending an itemized bill"
            modifier = { message ->
                narrationsGiven++
                "$message.\n(I have narrated $narrationsGiven things)"
            }
        }
        else -> {
            mood = "professional"
            modifier = { message ->
                "$message."
            }
        }
    }

    narrationModifier = modifier
    narrate("The narrator begins to feel $mood")
}
}
```

This new narrator mood will count the number of narrations given. Before you run your code, pause and ask yourself: What do you think this code will do? To confirm your suspicions, run NyetHack (it may take a couple of runs to get the new narrator mood). Your output will look like this:

```
A hero enters the town of Kronstadt. What is their name?
Madrigal
The narrator begins to feel like sending an itemized bill.
(I have narrated 1 things)
Madrigal, The Renowned Hero, heads to the town square.
(I have narrated 2 things)
```

Was your guess correct? Let's review what happened here.

Although the `narrationsGiven` variable is defined outside the lambda, the lambda has access to the variable and can modify it. Therefore, the `narrationsGiven` value increments from 0 to 1, and from 1 to 2.

Challenge: New Titles and Moods

NyetHack currently has five narrator moods and four titles that it can assign to players, but the sky is the limit. Get creative and add more narrator moods and player titles to your heart's content.

Here are some ideas you can use for narrator moods:

- **lazy**: The narrator will only say the first half of the message being narrated. (Hint: Take a look at **String**'s **take** or **substring** functions.)
- **leet** (or **1337**): The narrator will speak in leetspeak, replacing letters with numbers and symbols that look similar. For example, 'L' becomes '1'; 'E' becomes '3'; 'T' becomes '7'. (Hint: Take a look at **String**'s **replace** function. There is a version that accepts a lambda as the second parameter.)
- **poetic**: The narrator will insert line breaks between words in a way that looks vaguely like poetry. Unfortunately, because of the complexity of language and meter, the narrator will likely not be very good at this. (Hint: There are many solutions here, but one option is to combine the **replace** function with the **Random** class you saw earlier in the chapter.)

Likewise, here are some ideas for titles that you can assign to players:

- “The Bold”: Assigned to a player if all the letters in their name are capitalized.
- “The Verbose”: Assigned to a player if they have many letters in their name (“many” being whatever threshold you choose).
- “Bringer of Palindromes”: Assigned to a player if their name is a palindrome. (Hint: Take a look at **String**'s **reverse** function. Also, remember that strings are case sensitive.)

Lists and Sets

Working with groups of related values is an essential part of many programs. For example, your program might manage lists of books, travel destinations, menu items, or tavern patrons' tab balances. [Collections](#) allow you to conveniently work with those groups of values and pass them as arguments to functions.

You will see the most commonly used collection types in the next two chapters: **List**, **Set**, and **Map**. Like the other variable types you learned about in [Chapter 2](#), lists, sets, and maps come in two distinct varieties: mutable and read-only. In this chapter, we will focus on lists and sets.

Your NyetHack hero has just embarked on a long journey to reach the town of Kronstadt. To wind down and meet some citizens of this lively town, your hero's first destination is the local tavern, so in this chapter you will begin setting up this establishment. Collections will be an integral component of modeling NyetHack's tavern.

When your work is finished, the tavern will sport a full menu of items for purchase – along with a bustling scene of patrons eager to spend their gold. But before you can start building the tavern, there is some housekeeping to attend to. Create a new file called `Tavern.kt` and define a bit of scaffolding to support the tavern.

Listing 9.1 Laying the tavern's foundation (Tavern.kt)

```
private const val TAVERN_MASTER = "Taernyl"
private const val TAVERN_NAME = "$TAVERN_MASTER's Folly"

fun visitTavern() {
}
```

Next, it is time once again to update your **main** function in `NyetHack.kt`. You will insert a call to **visitTavern** at the end of **main**. And because you will need to use the hero's name throughout NyetHack, you will make it a top-level variable. Also, to prevent your **main** function from blowing up in size and complexity, you will extract the welcome message and input handling for the hero's name into its own function.

You will be running NyetHack several times in this chapter. Entering a hero name each time would be tedious, as would the antics of your moody narrator. To focus on the development, you will comment out your **readLine** implementation, instead assigning a default player name, as well as the call to **changeNarratorMood**.

Keep this code around, though. You will uncomment it in [Chapter 19](#) when you put the finishing touches on NyetHack.

Listing 9.2 Performing upkeep in main (NyetHack.kt)

```
var heroName: String = ""

fun main() {
narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
// Prints the message in yellow
"\u001b[33;1m$message\u001b[0m"
}

val heroName = readLine()
require(heroName != null && heroName.isNotEmpty()) {
"The hero must have a name."
}
heroName = promptHeroName()
changeNarratorMood()
// changeNarratorMood()
narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
visitTavern()
}

private fun promptHeroName(): String {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Prints the message in yellow
        "\u001b[33;1m$message\u001b[0m"
    }

    /*val input = readLine()
    require(input != null && input.isNotEmpty()) {
        "The hero must have a name."
    }

    return input*/
    println("Madrigal")
    return "Madrigal"
}
...
```

You have used a new way to comment out code here. The `//` that you are familiar with comments out a single line. `/*` and `*/` comment out all the code between them, even if it spans multiple lines. Although code comments of both styles are designed to add commentary about how your code behaves, it is also very common for developers to comment out code that they do not want to run temporarily.

Run NyetHack to check for errors. You should see this output:

```
A hero enters the town of Kronstadt. What is their name?
Madrigal
Madrigal, The Renowned Hero, heads to the town square
```


Now it is time to implement **visitTavern** so Taernyl's Folly can welcome its first customers.

Lists

Lists hold an ordered collection of values and allow duplicate values. They are the most prevalent collection type due to their relative ease of use and overall simplicity compared to other data structures. Because of their ordering properties in particular, lists are a great tool for storing information like a queue of tavern patrons or the list of items on the tavern menu.

In `Tavern.kt`, open your tavern for business by adding a list of patrons using the **listOf** function. **listOf** returns a read-only list populated with the elements you provide for the argument. Create your list with three patron names:

Listing 9.3 Creating a list of patrons (Tavern.kt)

```
private const val TAVERN_MASTER = "Taernyl"
private const val TAVERN_NAME = "$TAVERN_MASTER's Folly"

fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons: List<String> = listOf("Eli", "Mordoc", "Sophie")
    println(patrons)
}
```

Run NyetHack again. You should see:

```
A hero enters the town of Kronstadt. What is their name?
Madrigal
Madrigal, The Renowned Hero, heads to the town square
Madrigal enters Taernyl's Folly
[Eli, Mordoc, Sophie]
```

Up to now, you have been creating variables of various types by simply declaring them. But collections require two steps: creating the collection (here, the list to hold the patrons) and adding contents to it (the patron names). Kotlin provides functions like **listOf** that do both at once.

Now that you have a list, let's take a closer look at the **List** type.

Though type inference does work with lists, you included the type information – `val patrons: List<String>` – to make it visible for discussion. The diamond braces in `List<String>` indicate that it is a [parameterized type](#). It tells the compiler about the type that the contents of the list will be – in this case, **Strings**. Changing the type parameter changes what the compiler allows the list to hold.

If you tried to put an integer in the `patrons` list, the compiler would not allow it. Try adding a number to the list you defined:

Listing 9.4 Adding an integer to a list of strings (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons: List<String> = listOf("Eli", "Mordoc", "Sophie", 1)
    println(patrons)
}
```

IntelliJ warns you that the integer does not conform to the expected type, **String**. Type parameters are used with **List** because **List** is a [generic type](#). This means a list can hold any type of data, including textual data like strings (as in the case of `patrons`) or characters, numeric data like integers or doubles, or even a new type that you define. You will learn more about generics in [Chapter 18](#).

Undo your last change, either with IntelliJ's undo command (Command-Z [Ctrl-Z]) or by deleting the integer:

Listing 9.5 Correcting the list contents (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons: List<String> = listOf("Eli", "Mordoc", "Sophie"1)
    println(patrons)
}
```

Accessing a list's elements

You can access any element of a list using the element's [index](#) and the **get** function or – more commonly – the `[]` operator. Lists are [zero-indexed](#), so "Eli" is currently at index 0 in `patrons`, and "Sophie" is at index 2.

Change `Tavern.kt` to print only the first patron. Also, remove the explicit type information from `patrons` – now that you have seen the parameterized type that this **List** uses, you can return to using type inference for cleaner code.

Listing 9.6 Accessing the first patron (Tavern.kt)

```
...
```

```
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons: List<String> = listOf("Eli", "Mordoc", "Sophie")
    println(patrons[0])
}
```

Run NyetHack. You will see the first patron, `Eli`, printed.

You can also use **get** to access a list element, passing in the desired index. Calling `patrons.get(0)` works the same as calling `patrons[0]` – both calls return the first patron in the list.

Index boundaries and safe index access

Accessing an element by index requires care, because attempting to access an element at an index that does not exist – say, the fourth item from a list that contains only three – causes an **ArrayIndexOutOfBoundsException** exception.

Try this in the Kotlin REPL. (You can copy the first line from `Tavern.kt`.)

Listing 9.7 Accessing a nonexistent index (REPL)

```
val patrons = listOf("Eli", "Mordoc", "Sophie")
patrons[4]
```

The result is an **ArrayIndexOutOfBoundsException** indicating that index 4 is outside the bounds of the list.

Because accessing an element by an index can throw an exception, Kotlin provides safe index access functions that allow you to deal with the problem differently. For example, if you want to access the first or last element, **List** has convenience functions to do that:

```
patrons.first() // Eli
patrons.last() // Sophie
```

Other functions can specify an action to take instead of throwing an exception if the index is out of bounds. For example, one of these safe index access functions, **getOrElse**, takes two arguments: The first is the requested index (in parentheses, not square brackets). The second is a lambda that generates a default value, instead of an exception, if the requested index is not in the bounds of the list.

Try it out in the REPL:

Listing 9.8 Testing getOrElse (REPL)

```
patrons.getOrElse(4) { "Unknown Patron" }
"Unknown Patron"
```

This time, the result is `Unknown Patron`. Because the requested index was not present in the list, the lambda expression was called to obtain a fallback value.

Another safe index access function, **getOrNull**, returns `null` instead of throwing an exception. When you use **getOrNull**, you must decide what to do with the null value, as you saw in [Chapter 7](#). One option is to coalesce the null value to a default. Try using **getOrNull** with the null coalescing operator in the REPL.

Listing 9.9 Testing getOrNull (REPL)

```
patrons.getOrNull(4) ?: "Unknown Patron"
"Unknown Patron"
```

Again, the result is `Unknown Patron`.

Kotlin includes many “orNull” versions of functions on collections. For example, there are functions for `firstOrNull` and `lastOrNull`, which will return `null` if the list is empty. Be mindful of edge cases like this when using lists and other collections in your code.

Checking the contents of a list

The tavern has dark corners and secret back rooms. Fortunately, the keen-eyed tavern master keeps diligent records of which patrons have left or entered. If you ask whether a particular patron is present, the tavern master can tell you by looking at the `patrons` list.

Update `Tavern.kt` to use the **contains** function to check whether a particular patron is present:

Listing 9.10 Checking for a patron (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = listOf("Eli", "Mordoc", "Sophie")
    println(patrons[0])

    val eliMessage = if (patrons.contains("Eli")) {
        "$TAVERN_MASTER says: Eli's in the back playing cards"
    } else {
        "$TAVERN_MASTER says: Eli isn't here"
    }
}
```

```

    }
    println(eliMessage)
}

```

Run NyetHack. Because `patrons` does contain "Eli", you will see Taernyl says: Eli's in the back playing cards at the end of the console output.

By the way, the **contains** function performs a structural comparison for the elements in the list, like the structural equality operator. (Refer back to the comparison of structural and referential equality in [the section called *String Comparison*](#) in [Chapter 6](#) if you need a reminder of the difference.)

You can also use the **containsAll** function to check whether several patrons are present at once. To do this, you pass **containsAll** a list of the elements you want to check for. Update the code to ask the tavern master whether both Sophie and Mordoc are present:

Listing 9.11 Checking for multiple patrons (Tavern.kt)

```

...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = listOf("Eli", "Mordoc", "Sophie")

    val eliMessage = if (patrons.contains("Eli")) {
        "$TAVERN_MASTER says: Eli's in the back playing cards"
    } else {
        "$TAVERN_MASTER says: Eli isn't here"
    }
    println(eliMessage)

    val othersMessage = if (patrons.containsAll(listOf("Sophie", "Mordoc"))) {
        "$TAVERN_MASTER says: Sophie and Mordoc are seated by the stew kettle"
    } else {
        "$TAVERN_MASTER says: Sophie and Mordoc aren't with each other right now"
    }
    println(othersMessage)
}

```

Run NyetHack. You will see the following output:

```

...
Madrigal enters Taernyl's Folly
Taernyl says: Eli's in the back playing cards
Taernyl says: Sophie and Mordoc are seated by the stew kettle

```

Changing a list's contents

If a patron shows up or leaves, the watchful tavern master needs to add or remove the patron's name from the `patrons` variable. Currently, that is not possible.

listOf returns a read-only list that does not allow changes to its contents: You cannot add, remove, update, or replace entries. Read-only lists are a good idea, because they prevent unfortunate mistakes – like kicking a patron out into the cold by accidentally removing them from the list.

The read-only nature of the list has nothing to do with the `val` or `var` keyword you used to define the list variable. Changing the variable declaration for `patrons` from `val` (as it is defined now) to `var` would not change the list from read-only to writable. Instead, it would allow you to reassign the `patrons` variable to hold a new, different list.

List mutability is defined by the *type* of the list and refers to whether you can modify the elements in the list. Since patrons come and go from the tavern freely, the type of `patrons` needs to be changed to allow updates. In Kotlin, a modifiable list is known as a [mutable list](#). Its type is, unsurprisingly, **MutableList**, and you use the **mutableListOf** function to create one.

Update `Tavern.kt` to use **mutableListOf** instead of **listOf**. Mutable lists come with a variety of functions for adding, removing, and updating items. Simulate patrons coming and going by using the **add** and **remove** functions:

Listing 9.12 Making the patron list mutable (Tavern.kt)

```

...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = listOf("Eli", "Mordoc", "Sophie")
    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    narrate("Eli leaves the tavern")
    patrons.remove("Eli")
    narrate("Alex enters the tavern")
    patrons.add("Alex")
    println(patrons)
}

```

List also provides functions for moving between read-only and mutable versions on the fly: **toList** and **toMutableList**. For example, you could create a read-only version of the mutable `patrons` using **toList**:

```

val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
val readOnlypatrons = patrons.toList()

```

Run NyetHack. You will see the following printed to the console:

```

...

```

```
Eli leaves the tavern
Alex enters the tavern
[Mordoc, Sophie, Alex]
```

The new element, Alex, is added at the end of the list. You can also add a patron at a particular position in the list. For example, if a VIP comes into the tavern, the tavern master can prioritize their place in line.

Add a VIP patron – coincidentally also with the name Alex – to the beginning of the patron list. (This Alex is well-known around town and enjoys perks like being served first, much to the chagrin of the other Alex.) Lists support multiple elements with the same value, such as two patrons with the same name, so adding another Alex is no problem for the list.

Listing 9.13 Adding another Alex (Tavern.kt)

```
...
fun visitTavern() {
    ...
    narrate("Eli leaves the tavern")
    patrons.remove("Eli")
    narrate("Alex enters the tavern")
    patrons.add("Alex")
    println(patrons)
    narrate("Alex (VIP) enters the tavern")
    patrons.add(0, "Alex")
    println(patrons)
}

```

Run NyetHack again. You will see the following output:

```
...
[Mordoc, Sophie, Alex]
Alex (VIP) enters the tavern
[Alex, Mordoc, Sophie, Alex]
```

Say that the famous Alex would prefer to go by Alexis. Respect this wish by modifying patrons using the set operator ([]=) to reassign the string at the first index in the list.

Listing 9.14 Modifying a mutable list using the set operator (Tavern.kt)

```
...
fun visitTavern() {
    ...
    narrate("Eli leaves the tavern")
    patrons.remove("Eli")
    narrate("Alex enters the tavern")
    patrons.add("Alex")
    narrate("Alex (VIP) enters the tavern")
    patrons.add(0, "Alex")
    patrons[0] = "Alexis"
    println(patrons)
}

```

Run NyetHack. You will see that patrons has been updated with Alexis’s preferred name.

```
...
[Mordoc, Sophie, Alex]
Alex (VIP) enters the tavern
[Alexis, Mordoc, Sophie, Alex]
```

Functions that change the contents of a mutable list are called [mutator functions](#). [Table 9.1](#) lists the most commonly used mutator functions for lists.

Table 9.1 Mutable list mutator functions

Function	Description	Example(s)
[]= (set operator)	Sets the value at the index; throws an exception if the index does not exist.	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons[4] = "Reggie" IndexOutOfBoundsException</pre>
add	Adds an element to the end of the list, resizing the list by one element.	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.add("Reggie") [Eli, Mordoc, Sophie, Reggie] patrons.size 4</pre>
add (at index)	Adds an element to the list at a particular index, resizing the list by one element. Throws an exception if the index does not exist.	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.add(0, "Reggie") [Reggie, Eli, Mordoc, Sophie] patrons.add(5, "Sophie") IndexOutOfBoundsException</pre>
addAll	Adds another collection with contents of the same type to the list.	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.addAll(listOf("Reginald", "Alex")) [Eli, Mordoc, Sophie, Reginald, Alex]</pre>
+= (plus assign operator)	Adds an element or collection of elements to the list.	<pre>mutableListOf("Eli", "Mordoc", "Sophie") += "Reginald" [Eli, Mordoc, Sophie, Reginald] mutableListOf("Eli", "Mordoc", "Sophie") += listOf("Alex", "Shruti") [Eli, Mordoc, Sophie, Alex, Shruti]</pre>
-= (minus	Removes an element or collection of elements from the list.	<pre>mutableListOf("Eli", "Mordoc", "Sophie") -= "Eli" [Mordoc, Sophie]</pre>

assign operator)		<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons -= listOf("Eli", Mordoc) [Sophie]</pre>
clear	Removes all the elements from the list.	<pre>mutableListOf("Eli", "Mordoc", Sophie").clear() []</pre>
remove	Removes an element from the list.	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.remove("Mordoc") [Eli, Sophie]</pre>
removeAll	Removes all elements in another collection from the list.	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.removeAll(listOf("Eli", "Mordoc") [Sophie]</pre>

Iteration

The tavern master makes a point of greeting each patron, as it is good business to do so. Lists include built-in support for a variety of functions that allow you to perform an action for each element of their contents. This concept is called [iteration](#).

One way to iterate through a list is a `for` loop. Its logic is, “for each element in the list, do something.” You give the element a name, and the Kotlin compiler will automatically detect its type for you.

Update `Tavern.kt` to print a greeting for each patron. (Also, remove the code from earlier that modifies and prints `patrons` to tidy up your console output.)

Listing 9.15 Iterating over the patrons with `for` (`Tavern.kt`)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    narrate("Eli leaves the tavern")
    patrons.remove("Eli")
    narrate("Alex enters the tavern")
    patrons.add("Alex")
    println(patrons)
    narrate("Alex (VIP) enters the tavern")
    patrons.add(0, "Alex")
    patrons[0] = "Alexis"
    println(patrons)

    for (patron in patrons) {
        println("Good evening, $patron")
    }
}
```

Run `NyetHack`, and the tavern master will greet each patron by name:

```
...
Good evening, Eli
Good evening, Mordoc
Good evening, Sophie
```

In this case, because `patrons` is of type **`MutableList<String>`**, `patron` will be of type **`String`**. Within the block of the `for` loop, any code that you apply to `patron` will be applied to each of the elements in `patrons`.

Note the `in` keyword:

```
for (patron in patrons) { ... }
```

`in` specifies the object being iterated over in a `for` loop.

In many languages, like Java, C#, and JavaScript, this loop syntax is called a `foreach` loop. You may be familiar with another loop syntax, which commonly appears as

```
for (int i = 0; i < patrons.size; i++)
```

in Java and C code. Kotlin does not support this syntax. The only type of `for` loop that Kotlin supports is the `foreach` loop. If you do need to iterate over a specific set of numbers, you can use the **`Range`** type you saw in [Chapter 3](#). Ranges are customizable and support many of the same behaviors that you can accomplish with a traditional `for` loop.

```
// Prints every patron
for (i in 0 until patrons.size) {
    println(patrons[i])
}

// Prints every other patron in reverse order
for (i in patrons.size - 1 downTo 0 step 2) {
    println(patrons[i])
}
```

You may be wondering: Is there a performance penalty to using this kind of loop? Fear not. When your Kotlin code is compiled, the compiler will optimize your `foreach` loops into traditional `for` loops. The compiler is also smart enough to implement **`Range`**-based `foreach` loops without creating an instance of a **`Range`** at runtime.

The `for` loop is simple and readable, but there is another option: **`List`** and **`MutableList`** also have a function called **`forEach`**.

The **`forEach`** function traverses each element in the list – one by one, from left to right – and passes each element to the lambda expression you provide as an argument.

Replace your `for` loop with the **`forEach`** function.

Listing 9.16 Iterating over the patrons with `forEach` (`Tavern.kt`)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    for (patron in patrons) {
```



```

        println("Good evening, $patron")
    }
    patrons.forEach { patron ->
        println("Good evening, $patron")
    }
}

```

Run NyetHack, and you will see the same output as before. The for loop and the **forEach** function are functionally equivalent.

Kotlin’s for loop and **forEach** function handle indexing behind the scenes. If you also want access to the index of each element in a list as you iterate, use **forEachIndexed**. Update Tavern.kt to use **forEachIndexed** to display each patron’s position in line:

Listing 9.17 Displaying line position with forEachIndexed (Tavern.kt)

```

... visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    patrons.forEach { patron ->
    patrons.forEachIndexed { index, patron ->
        println("Good evening, $patron - you're #${index + 1} in line")
    }
}

```

Run NyetHack again to see the patrons and their positions:

```

...
Good evening, Eli - you're #1 in line
Good evening, Mordoc - you're #2 in line
Good evening, Sophie - you're #3 in line

```

In general, we prefer the **forEach** function over for loops. We find that it tends to be more readable and feels right at home when combined with the functional programming operations that you will see in [Chapter 11](#). We also find that when iteration logic requires the index of an item in the list, **forEachIndexed** is the cleanest way to go.

The **forEach** and **forEachIndexed** functions are also available on certain other types in Kotlin. This category of types is called **Iterable**, and **List**, **Set**, **Map**, **IntRange** (ranges like 0..9, which you saw in [Chapter 3](#)), and other collection types belong to the **Iterable** category. An iterable supports iteration – in other words, it allows traversing the elements it holds, performing some action for each element.

Some other types in Kotlin – such as **String** – also support these common iteration functions despite not being implemented as an **Iterable** type. Keep an eye out for these kinds of functions. You will see in [Chapter 11](#) that there are some powerful techniques often used with **Iterable** types to manipulate data in your code.

With Taernyl greeting guests, he is now ready to accept orders. Have each patron place an order for a local favorite called Dragon’s Breath. To do so, create a new function called **placeOrder** and call it from the lambda expression that you passed to the **forEachIndexed** function. This will cause all patrons in the list to place an order.

Listing 9.18 Simulating several orders (Tavern.kt)

```

...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    patrons.forEachIndexed { index, patron ->
        println("Good evening, $patron - you're #${index + 1} in line")
        placeOrder(patron, "Dragon's Breath")
    }
}

private fun placeOrder(patronName: String, menuItemName: String) {
    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    narrate("$TAVERN_MASTER hands $patronName a $menuItemName")
}

```

Run NyetHack and watch the tavern spring to life as the three patrons excitedly place their orders for Dragon’s Breath:

```

...
Good evening, Eli - you're #1 in line
Eli speaks with Taernyl to place an order
Taernyl hands Eli a Dragon's Breath
Good evening, Mordoc - you're #2 in line
Mordoc speaks with Taernyl to place an order
Taernyl hands Mordoc a Dragon's Breath
Good evening, Sophie - you're #3 in line
Sophie speaks with Taernyl to place an order
Taernyl hands Sophie a Dragon's Breath

```

Iterable collections support a variety of functions that let you define an action to perform for each item in the collection. You will learn more about **Iterables** and the other iteration functions in [Chapter 11](#).

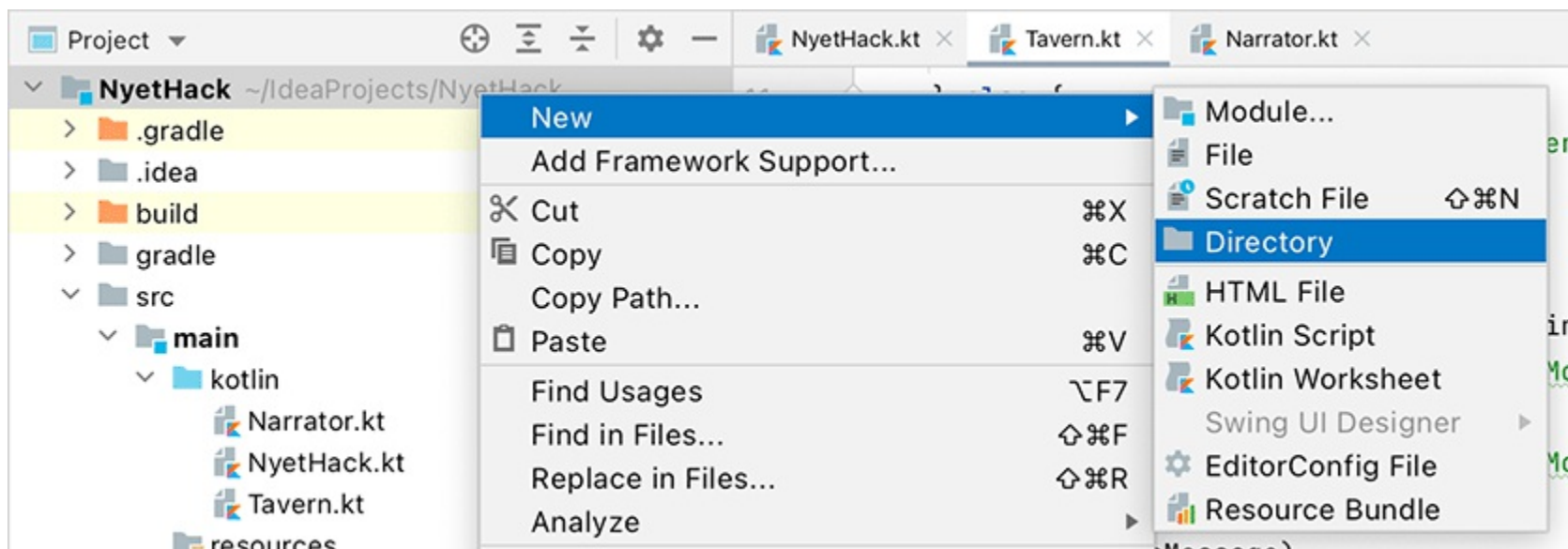
Reading a File into a List

Variety is the spice of life, and the tavern master knows that patrons expect a variety of items on the menu. Currently, Dragon's Breath is the only item for sale. Time to fix that by loading some menu items for patrons to choose from.

To save you some typing, we will provide you with predefined menu data in a text file you can load into NyetHack. The file contains several menu items and some additional metadata for each item that you will utilize in the next chapter.

Start by creating a new folder for data: In the project tool window, right-click the NyetHack project and choose New → Directory ([Figure 9.1](#)). Name the directory `data`.

Figure 9.1 Creating a new directory



Next, download the menu data from bignerdranch.com/tavern-menu-data/ and save it to the `data` folder you created in a file called `tavern-menu-data.txt`.

Now you can update `Tavern.kt` to read the text from that file into a string. To make the menu easier to work with, you will use the `split` function. `split` returns a `List<String>` where each entry corresponds to a section of the `String` that appears between a delimiter (in this case, the newline character).

When you make this change, be sure to include the `import java.io.File` statement at the very top of `Tavern.kt`. (Also keep in mind that because `File` is a Java API, you can only use it when targeting Kotlin/JVM. For other targets, you will need to pick an API appropriate for the platform you are compiling for.)

Listing 9.19 Reading menu data from a file (`Tavern.kt`)

```
import java.io.File

private const val TAVERN_MASTER = "Taernyl"
private const val TAVERN_NAME = "$TAVERN_MASTER's Folly"

private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")
...

```

You used the `java.io.File` type to work with a particular file by providing a file path.

The `readText` function on `File` returns the contents of the file as a `String`. Then you use the `split` function to return a list, splitting on the newline character (represented by the escape sequence `'\n'`).

Now, call `forEachIndexed` on `menuData` to print out each entry in the `List` along with its index.

Listing 9.20 Printing the diversified menu (`Tavern.kt`)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    patrons.forEachIndexed { index, patron ->
        println("Good evening, $patron - you're #${index + 1} in line")
        placeOrder(patron, "Dragon's Breath")
    }

    menuData.forEachIndexed { index, data ->
        println("$index : $data")
    }
}
...

```

Run `NyetHack`. You will see the menu data that was loaded into the `List`:

```
...
0 : shandy,Dragon's Breath,5.91

```

1 : elixir,Shirley's Temple,4.12
2 : meal,Goblet of LaCroix,1.22
3 : desert dessert,Pickled Camel Hump,7.33
4 : elixir,Iced Boilermaker,11.22
5 : deserved dessert,Hard Day's Work Ice Cream,3.21
6 : meal,Bite of Lembas Bread,0.59

Destructuring

The `menuData` list contains some additional metadata about the items on the menu, including what kind of menu item each one is and how much it costs. This information will come in handy when you need to purchase these items (which you will do in [Chapter 10](#)), but for now you only need the items’ names.

Create a second list to store the item names. There are two new pieces of syntax you will use to accomplish this, which we will explain after you enter this code.

Listing 9.21 Parsing menu item names (Tavern.kt)

```
...
private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")

private val menuItems = List(menuData.size) { index ->
    val (type, name, price) = menuData[index].split(",")
    name
}

fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    println(menuItems)

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
menuData.forEachIndexed { index, data ->
    println("$index : $data")
}
}
```

To create the `menuItems` list, you are using the **List** *constructor* instead of the **listOf** function. You will learn more about constructors in [Chapter 13](#), but for now you can treat it as a regular function that returns a **List**. The **List** constructor accepts two arguments: an **Int** size and an initialization function to populate the list.

Take a look at the lambda expression you provide to the **List** constructor. First, it splits one of the lines from the menu items file everywhere a comma appears. The result of this split is a list that looks like this:

```
["shandy", "Dragon's Breath", "5.91"]
```

To populate your `menuItems` list, you want the second item in this list ("Dragon's Breath" in this case). While you could use the get operations you saw before to get the item at index 1, here you are using a different technique, called *destructuring*.

A list offers the ability to *destructure* up to the first five elements it contains. Destructuring allows you to declare and assign multiple variables in a single expression. You are using this destructuring declaration to separate the elements of the menu data:

```
val (type, name, price) = menuData[index].split(',')
```

This declaration assigns the first three elements in the list returned by the **split** function to string values named `type`, `name`, and `price`.

Run `NyetHack` again. The menu items’ names will now be printed as a list instead of a numbered sequence:

```
...
Madrigal enters Taernyl's Folly
There are several items for sale:
[Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
 Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread]
...
```

You can also selectively destructure elements from a list by using the symbol `_` to skip unwanted elements. Because your list initializer does not need to know about the item’s type or price, omit them using `_`:

Listing 9.22 Omitting destructured elements (Tavern.kt)

```
...
private val menuItems = List(menuData.size) { index ->
val (type, name, price) = menuData[index].split(",")
    val (_, name, _) = menuData[index].split(",")
    name
}
...
```

Run `NyetHack` again and confirm that your output did not change.

Now that the menu data is loaded and you have a quick reference to the items on the menu, have each patron choose randomly from the menu when placing their order:

Listing 9.23 Placing random orders (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    println(menuItems)

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    patrons.forEachIndexed { index, patron ->
        println("Good evening, $patron - you're #${index + 1} in line")
        placeOrder(patron, "Dragon's Breath")
    }
}
```

```
        placeOrder(patron, menuItems.random())
    }
    ...
}
```

Run NyetHack. You will see each patron place an order for a random item on the menu.

Sets

Lists, as you have seen, allow duplicate elements (and are ordered, so duplicates – and other elements – can be identified by their position). But sometimes you want a collection that guarantees that its items are unique. For that, you use a **Set**.

Sets are like **Lists** in many ways. They use the same iteration functions, and **Set** also comes in read-only and mutable flavors.

But there are two major differences between lists and sets: The elements of a set are unique, and sets are not index-based, because the items in a set are not guaranteed to be in any particular order. (That said, you can still read an element at a particular index, which we will discuss shortly.)

Creating a set

Just as you create a list using the **listOf** function, you create a **Set** using the **setOf** function. Try creating a set in the REPL:

Listing 9.24 Creating a set (REPL)

```
val planets = setOf("Mercury", "Venus", "Earth")
planets
["Mercury", "Venus", "Earth"]
```

If you try to create the planets set with a duplicate, only one of the duplicate items will remain in the set:

Listing 9.25 Trying to create a set with a duplicate (REPL)

```
val planets = setOf("Mercury", "Venus", "Earth", "Earth")
planets
["Mercury", "Venus", "Earth"]
```

The duplicate element "Earth" was dropped from the set.

As with a list, you can check whether a set contains a particular element using **contains** and **containsAll**. You can also use the **in** keyword as a shorthand for calling the **contains** function (there is no shorthand for **containsAll**). Try the **contains** function in the REPL:

Listing 9.26 Checking planets (REPL)

```
planets.contains("Pluto")
false

"Earth" in planets
true
```

Set does not index its contents – meaning it provides no built-in `[]` operator to access elements using an index. However, you can still request an element at a particular index, using functions that use iteration to accomplish the task. Enter the code below into the REPL to read the third planet in the set with the **elementAt** function:

Listing 9.27 Finding the third planet (REPL)

```
planets.elementAt(2)
Earth
```

While this works, using index-based access with a set is an order of magnitude slower than index-based access with a list, because of the way **elementAt** works under the hood. When you call the **elementAt** function on the set, the set iterates to the index you provide, one element at a time. This means that for a large set, requesting an element at a high index would be slower than accessing an element by index in a list. For this reason, if you want index-based access, you probably want a **List**, not a **Set**.

Also, while **Set** does have a mutable version (which you will soon see), no mutator functions are available that rely on indices (like **List**'s **add(index, element)** function).

To make up for this performance deficiency, sets come with a unique advantage over lists: checking whether or not an item is in a set is extremely fast, regardless of how large the set becomes. Behind the scenes, **Set** has an internal ordering of items that lets it find elements very quickly.

Consider your tavern. A list works well for keeping track of the order patrons are lined up in to be served. But if you simply want to keep track of who is in the tavern, a set is likely preferable – as long as your patrons have unique names.

Adding elements to a set

Taarnyl would like to remove the queueing system from his tavern, but he still needs to know who is there. For your

next task, update NyetHack’s tavern to store the tavern’s patrons in a **MutableSet** instead of a **MutableList**. Because many people can have the same first name, Taernyl will need to track the full names of his patrons.

To accomplish this in NyetHack, randomly generate patron names. Update `Tavern.kt` with a collection of first and last names, and use **repeat** to generate 10 random combinations of names. Add the generated patron names to your new set using the `+=` operator. (You could also use the **add** function, but we have used the operator in this case.)

Remove the two calls to **forEachIndexed** that created patron greetings and menu orders. Print out the names of people in the tavern; to make this output more reader-friendly, use the **joinToString** function, which concatenates all items in a collection and separates them with commas. Finally, have three random patrons place their orders from the menu.

Listing 9.28 Generating random patrons (Tavern.kt)

```
private const val TAVERN_MASTER = "Taernyl"
private const val TAVERN_NAME = "$TAVERN_MASTER's Folly"

private val firstNames = setOf("Alex", "Mordoc", "Sophie", "Tariq")
private val lastNames = setOf("Ironfoot", "Fernsworth", "Baggins", "Downstrider")

private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    println(menuItems)
    narrate(menuItems.joinToString())

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    val patrons: MutableSet<String> = mutableSetOf()
    repeat(10) {
        patrons += "${firstNames.random()} ${lastNames.random()}"
    }

    val eliMessage = if (patrons.contains("Eli")) {
        "$TAVERN_MASTER says: Eli's in the back playing cards"
    } else {
        "$TAVERN_MASTER says: Eli isn't here"
    }
    println(eliMessage)

    val othersMessage = if (patrons.containsAll(listOf("Sophie", "Mordoc"))) {
        "$TAVERN_MASTER says: Sophie and Mordoc are seated by the stew kettle"
    } else {
        "$TAVERN_MASTER says: Sophie and Mordoc aren't with each other right now"
    }
    println(othersMessage)

    patrons.forEachIndexed { index, patron ->
        println("Good evening, $patron - you're #${index + 1} in line")
        placeOrder(patron, menuItems.random())
    }

    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())

    repeat(3) {
        placeOrder(patrons.random(), menuItems.random())
    }
}
...
```

Run NyetHack. You will see random patron names in the output. They will not necessarily match the ones below, but they will be similar:

```
...
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
Madrigal sees several patrons in the tavern:
Alex Downstrider, Sophie Downstrider, Mordoc Fernsworth, Tariq Downstrider,
Mordoc Baggins, Mordoc Ironfoot, Alex Fernsworth, Alex Baggins,
Tariq Fernsworth, Tariq Baggins
Alex Fernsworth speaks with Taernyl to place an order
Taernyl hands Alex Fernsworth a Shirley's Temple
Mordoc Ironfoot speaks with Taernyl to place an order
Taernyl hands Mordoc Ironfoot a Hard Day's Work Ice Cream
Alex Downstrider speaks with Taernyl to place an order
Taernyl hands Alex Downstrider a Hard Day's Work Ice Cream
```

As we mentioned earlier, while **MutableSet** supports adding and removing elements, like **MutableList**, it does not provide index-based mutator functions. [Table 9.2](#) shows some of the most commonly used **MutableSet** mutator functions.

Table 9.2 Mutable set mutator functions

Function	Description	Example(s)
add	Adds an element to the set.	<code>mutableSetOf(1, 2).add(3)</code> <code>[1, 2, 3]</code>
addAll	Adds all of another collection with contents of the same type to the set.	<code>mutableSetOf(1, 2).addAll(listOf(1, 5, 6))</code> <code>[1, 2, 5, 6]</code>
+= (plus assign operator)	Adds an element or collection of elements to the set.	<code>mutableSetOf(1, 2) += 3</code> <code>[1, 2, 3]</code> <code>mutableSetOf(1, 2) += listOf(1, 3, 5, 5)</code> <code>[1, 2, 3, 5]</code>
-= (minus assign operator)	Removes an element or collection of elements from the set.	<code>mutableSetOf(1, 2, 3) -= 3</code> <code>[1, 2]</code>

operator)		<code>mutableSetOf(1, 2, 3) -= listOf(2, 3)</code> <code>[1]</code>
remove	Removes the element from the set.	<code>mutableSetOf(1, 2, 3).remove(1)</code> <code>[2, 3]</code>
removeAll	Removes all elements in another collection from the set.	<code>mutableSetOf(1, 2).removeAll(listOf(1, 3))</code> <code>[2]</code>
clear	Removes all elements from the set.	<code>mutableSetOf(1, 2).clear()</code> <code>[]</code>

Run NyetHack several more times, paying close attention to the patrons in the tavern. You might see that the tavern sometimes gets populated with fewer than 10 patrons. This happens when a name is generated twice, since sets do not allow duplicates.

while Loops

If you want to ensure that there are always 10 patrons in the tavern when Madrigal enters, you will need a different approach to populate your set. There is another control flow mechanism that is perfect for this task: a `while` loop.

`for` loops are a useful form of control flow when you want to run some code for each element in series. But they are not as good at representing state that cannot be iterated through. That is where `while` loops are useful.

A `while` loop’s logic is, “While some condition is true, execute the code in this block.” Update `Tavern.kt` to generate patrons by using a `while` loop until exactly 10 entries are in the set.

Listing 9.29 Always generating 10 patrons (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    narrate(menuItems.joinToString())

    val patrons: MutableSet<String> = mutableSetOf()
    repeat(10) {
    while (patrons.size < 10) {
        patrons += "${firstNames.random()} ${lastNames.random()}"
    }
    ...
}
...
```

Run `NyetHack` several more times. This time, you will consistently see 10 patrons in the tavern.

A `while` loop requires that you have some foundation in place to keep track of the loop’s state. Here, you are taking advantage of the fact that `patrons` automatically keeps track of its size. Depending on your goal, you can also incorporate your own counters or conditions that you manage manually.

You can represent more complex state by combining `while` loops with other forms of control flow, like the conditionals you saw in [Chapter 3](#). Consider this example:

```
var isTavernOpen = true
var isClosingTime = false
while (isTavernOpen) {
    if (isClosingTime) {
        isTavernOpen = false
    }

    println("Having a grand old time!")
}
```

Here, the `while` loop continues to loop as long as `isTavernOpen` is true, keeping track of state represented by a **Boolean**.

`while` loops are powerful – but they can also be dangerous. Consider what would happen if `isTavernOpen` was never false. Or if you wanted the tavern to hold 20 people, when there are only 16 unique names that can be randomly generated. In both of these cases, the `while` loop would loop forever, and the program would “hang,” or continue to execute indefinitely. Take care when using `while` loops for this reason.

Collection Conversion

In NyetHack, you create a mutable set of unique patron names by feeding the elements from a list into it, one by one. You can also convert a list to a set, or vice versa, using the **toSet** and **toList** functions (or their mutable cousins: **toMutableSet** and **toMutableList**). A common trick is to call **toSet** to drop the non-unique elements in a list. Try these experiments in the REPL.

Listing 9.30 Converting a list to a set (REPL)

```
listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot").toSet()
[Eli Baggins, Eli Ironfoot]
```

If you want quick index-based access after converting a list to a set to remove duplicates, you can convert the set back to a list:

Listing 9.31 Converting a set back to a list (REPL)

```
val patrons = listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot")
    .toSet()
    .toList()
patrons
[Eli Baggins, Eli Ironfoot]

patrons[0]
Eli Baggins
```

The need to remove duplicates and resume index-based access is so common that Kotlin provides a function on **List** called **distinct** that behaves the same way:

Listing 9.32 Calling distinct (REPL)

```
val patrons = listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot").distinct()
patrons
[Eli Baggins, Eli Ironfoot]

patrons[0]
Eli Baggins
```

Sets are useful for representing series of data where each element is unique. In the next chapter, you will complete your tour of the Kotlin collection types by learning about maps as you finish the tavern simulation.

For the More Curious: Array Types

Many programming languages – including Kotlin – support primitive definitions of **Arrays**. Arrays are much more basic than the collection types you have seen so far: They do not support resizing, are always mutable, and overwrite values in the array instead of making room for them.

Many times – especially when using platform code – you will be required to provide an **Array** instead of a collection.

Suppose you were attempting to call a function that had this signature in Kotlin:

```
fun displayPlayerAges(playerAges: IntArray)
```

The parameter expected by this **displayPlayerAges** is **IntArray**. In other programming languages like C and Java, this often appears as `int[] playerAges`. To obtain an **IntArray** (which will compile into an `int[]`), you can call **displayPlayerAges**, like this:

```
val playerAges: IntArray = intArrayOf(34, 27, 14, 52, 101)
displayPlayerAges(playerAges)
```

Notice the **IntArray** type and the **intArrayOf** function that was called. Like a **List**, an **IntArray** represents a series of elements – specifically integers. Unlike a **List**, an **IntArray** is backed by a primitive type when compiled. Kotlin will use the native array type for the platform you are targeting, making this a helpful technique to be aware of if you plan to use a lot of interoperability in your program.

It is also possible to convert a Kotlin collection to the required arrays using built-in conversion functions. For example, you could convert a list of integers to an **IntArray** using the **toIntArray** function provided by **List**. This would allow you to convert a collection to a primitive **int** array, which can be very helpful when calling Java or native functions:

```
val playerAges: List<Int> = listOf(34, 27, 14, 52, 101)
displayPlayerAges(playerAges.toIntArray())
```

[Table 9.3](#) shows the array types and the functions that create them.

Table 9.3 Array types

Array type	Creation function
IntArray	intArrayOf
DoubleArray	doubleArrayOf
LongArray	longArrayOf
ShortArray	shortArrayOf
ByteArray	byteArrayOf
FloatArray	floatArrayOf
BooleanArray	booleanArrayOf
Array ^a	arrayOf
^a Array compiles to a primitive array that holds any reference type.	

As a general rule, stick with the collection types like **List** unless you have a compelling reason to do otherwise – like the need to interoperate with platform code. A Kotlin collection is a better choice in most cases because collections provide the concept of “read-only-ness” versus mutability and support a more robust set of features.

For the More Curious: Read-Only vs Immutable

Throughout this book, we have favored the term “read-only” over “immutable,” with few exceptions – but we have not explained why. Now is the time. “Immutable” means “unchangeable,” and we think it is a misleading label for Kotlin collections (and certain other types) because they can, indeed, change. Let’s look at some examples using lists.

Here are declarations of two **List**s. They are read-only – declared with `val`. The element each one happens to contain is a mutable list.

```
val x = listOf(mutableListOf(1, 2, 3))
val y = listOf(mutableListOf(1, 2, 3))

x == y
true
```

So far, so good.

`x` and `y` were assigned with the same value, and the **List** API does not expose any functions for adding, removing, or reassigning a particular element. However, the lists contain mutable lists, and *their* contents *can* be modified:

```
val x = listOf(mutableListOf(1, 2, 3))
val y = listOf(mutableListOf(1, 2, 3))
x[0].add(4)

x == y
false
```

The structural comparison between `x` and `y` now evaluates as false, because the contents of `x` mutated. Should an immutable (“unchangeable”) list behave this way? In our opinion, it should not.

Here is another example:

```
var myList: List<Int> = listOf(1, 2, 3)
(myList as MutableList)[2] = 1000
myList
[1, 2, 1000]
```

In this example, `myList` was cast to the **MutableList** type – meaning that the compiler was instructed to treat `myList` as a mutable list, despite the fact that it was created with `listOf`. (You will read about casting in depth in [Chapter 15](#) and [Chapter 17](#).) This cast has the effect of allowing a change to the value of the third item in `myList`. Again, not the behavior we expect of something labeled “unchangeable.”

A **List** in Kotlin does not enforce immutability – it is up to you to use it in an immutable fashion. A Kotlin **List**’s “immutability” is only skin deep – and whatever you wind up calling it, remember that.

For the More Curious: The break Expression

`for` and `while` loops both exit when they reach a certain state. For a `for` loop, that is when there are no more items to iterate over. For a `while` loop, that happens when the condition given to the `while` loop evaluates to false.

Another way to exit a loop is to use the `break` expression. Consider the example below, in which a `while` loop runs while the tavern is open. Instead of changing `isTavernOpen`'s value to false to end the loop, a `break` expression would halt the loop immediately:

```
var isTavernOpen = true
var isClosingTime = false
while (isTavernOpen) {
    if (isClosingTime) {
        break
    }

    println("Having a grand old time!")
}
```

`break` does not stop execution of your program or function. Rather, it simply breaks out of the loop it is called from, and program execution continues.

`break` can be an easy way to exit a loop, but it often comes at the cost of readability. `break` in particular can be confusing when used with nested loops – which loop is being broken out of? (The answer to this question is that the innermost loop that the `break` is a child of will be stopped. But readers of your code who do not know this fact may be left scratching their heads.)

Complex loops in particular are often easier to read when all their looping logic appears in the loop condition itself. Otherwise, you might need to read the entire body of the loop to figure out its true meaning. For the previous example, you could express the same loop like this:

```
var isTavernOpen = true
var isClosingTime = false
while (isTavernOpen && !isClosingTime) {
    println("Having a grand old time!")
}
```

You will occasionally see `break` used in the Kotlin standard library or in other libraries that you add to your project, but we recommend thinking twice before using one in your own code.

For the More Curious: Return Labels

In [Chapter 8](#), you saw all the different ways to use lambdas. You can pass function references around or return them from other functions, but most often you will use lambdas by calling into Kotlin standard library functions like **forEach**.

Lambdas return a result implicitly, without a return keyword. You *can* include the return keyword within a lambda expression, but the result may surprise you. Consider a function that prints each letter in the English alphabet, skipping the vowels:

```
fun printConsonants() {
    ('a'..'z').forEach { letter ->
        if ("aeiou".contains(letter)) {
            return
        }
        print(letter)
    }
}
```

What is the result when you run **printConsonants**? (You can try it out in the Kotlin REPL.)

Rather than skipping an iteration, the return expression exits the **printConsonants** function. Although you called **return** from within the lambda expression, it returns from the outer scope, **printConsonants**. Because “a” is a vowel and the first letter of the English alphabet, **printConsonants** returns before anything can be printed out.

If you want to return out of the lambda expression passed to **forEach**, you have to be more explicit about where you are returning to. *Return labels* provide that specificity.

```
fun printConsonants() {
    ('a'..'z').forEach letters@{ letter ->
        if ("aeiou".contains(letter)) {
            return@letters
        }
        print(letter)
    }
}
```

In this modified version of the **printConsonants** function, the lambda expression passed to **forEach** has been given a label, **letters@**. **letters** is the name of the label, and the @ character denotes that this is a label that applies to the lambda expression it prepends.

When you add a label to your return statement, you can return to any of your local function scopes. Here, **return@letters** simply moves into the next iteration of a **forEach** loop, but you can also use labels to return values:

```
return@numbers 17
```

Labels can also be applied to loops, which can bring clarity to the break statement. Consider this code, which uses nested loops to print out all possible version numbers of a particular piece of software:

```
prefixLoop@for(prefix in listOf("alpha", "beta")) {
    var number = 0
    numbersLoop@while (number < 10) {
        val identifier = "$prefix $number"
        if (identifier == "beta 3") {
            break@prefixLoop
        }
        number++
    }
}
```

Take care when using labels, as changing scopes without context may be confusing. With proper care, however, labels can be another useful tool for controlling execution flow.

Challenge: Formatted Tavern Menu

First impressions go a long way, and one of the first things a patron will see is the tavern menu. For this challenge, generate a more elegant version of the menu to kick it up a notch. Show the menu as a uniformly aligned list of items and prices. The prices should be aligned by their decimal points. The whole menu should be presented in a pleasing block.

Your output should resemble this example:

```
*** Welcome to Taernyl's Folly ***

Dragon's Breath.....5.91
Shirley's Temple.....4.12
Goblet of LaCroix.....1.22
Pickled Camel Hump.....7.33
Iced Boilermaker.....11.22
Hard Day's Work Ice Cream.....3.21
Bite of Lembas Bread.....0.59
```

Hint: You will need to calculate the amount of padding for each line by using the longest string from the list of menu items.

Challenge: Advanced Formatted Tavern Menu

Building on the previous menu formatting code, generate a menu that additionally groups the elements to be listed by their type. The output should resemble the following:

```
*** Welcome to Taernyl's Folly ***
~[shandy]~
Dragon's Breath.....5.91
~[elixir]~
Iced Boilermaker.....11.22
Shirley's Temple.....4.12
~[meal]~
Goblet of LaCroix.....1.22
Bite of Lembas Bread.....0.59
~[desert dessert]~
Pickled Camel Hump.....7.33
~[deserved dessert]~
Hard Day's Work Ice Cream.....3.21
```

10

Maps

The third commonly used type of collection in Kotlin is **Map**. The **Map** type has a lot in common with the **List** and **Set** types: All three group elements, are read-only by default, use parameterized types to tell the compiler the type of their contents, and support iteration.

Where **Map** is different from **List** and **Set** is that its elements consist of key-value pairs, and instead of index-based access using an integer, a map provides key-based access using a type that you specify. Keys are unique and identify the values in the map; values, on the other hand, do not need to be unique. In this way, **Map** shares another feature with **Set**: The keys of a map are guaranteed to be unique, just like the elements of a set.

Creating a Map

Like lists and sets, maps are created using functions: **mapOf** and **mutableMapOf**. You will be using **mapOf** to create a map representing the amount of gold in various characters’ purses. For now, you will only track balances for Madrigal and Taernyl – you will come back to handle your randomly generated patrons later in this chapter.

Create your first map in `Tavern.kt`. (We will explain the argument syntax shortly.)

Listing 10.1 Creating a read-only map (`Tavern.kt`)

```
... visitTavern() {
    ...
    val patrons: MutableSet<String> = mutableSetOf()
    val patronGold = mapOf(
        TAVERN_MASTER to 86.00,
        heroName to 4.50
    )
    while (patrons.size < 10) {
        patrons += "${firstNames.random()} ${lastNames.random()}"
    }

    println(patronGold)

    narrate("$heroName sees several patrons in the tavern:")
    ...
}
```

While the keys in a map must all be of the same type, and the values must be of the same type, the keys and values can be of different types. Here you have a map with string keys and double values. You are using type inference, but if you had wanted to include explicit type information, it would look like this: `val patronGold: Map<String, Double>`.

Run `NyetHack` to see the map printed.

```
...
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
{Taernyl=86.0, Madrigal=4.5}
Madrigal sees several patrons in the tavern:
...
```

When a map is printed, it is shown in curly braces, while lists and sets are both shown in square brackets.

You used **to** to define each entry (key and value) in the map:

```
...
val patronGold = mapOf(
    TAVERN_MASTER to 86.00,
    heroName to 4.50
)
```

to may look like a keyword, but in fact it is a special type of function called an *[infix function](#)* that allows you to drop the dot and the parentheses around its argument. You could have written this as `heroName.to(4.50)`, but the shorthand is preferred – especially when creating maps using the **mapOf** function. You will learn more about this style of function call in [Chapter 19](#). The **to** function converts the values on its lefthand and righthand sides into a **Pair** – a type for representing a group of two elements.

Maps are built using key-value **Pairs**. In fact, another way you could have defined the entries for the map is as follows. (Try it in the REPL.)

Listing 10.2 Defining a map using the **Pair** type (REPL)

```
val patronGold = mapOf(
    Pair("Taernyl", 86.00),
    Pair("Madrigal", 4.50)
)
println(patronGold)
{Taernyl=86.0, Madrigal=4.5}
```

However, building a map using the **to** function is cleaner than this syntax.

We have said that the keys in a map must be unique. What if you tried adding a duplicate entry to the map? In the REPL, add another pair with "Madrigal" for the key:

Listing 10.3 Attempting to add a duplicate key (REPL)

```
val patronGold = mapOf(
    "Taernyl" to 86.00,
    "Madrigal" to 4.50,
    "Madrigal" to 20.00
)
println(patronGold)
{Taernyl=86.0, Madrigal=20.0}
```

Much like how sets require all elements to be unique, the keys in a map must be unique. If you attempt to add another **Pair** whose key already exists in the map, the original mapping will be replaced with the new key-value pair.

Accessing Map Values

You access a value in a map using its key. Much like how you accessed items from a list, you can use the **get** function or – more commonly – the indexing operator (`[]`). Instead of passing in an index, you provide the key that you want to look up the value for. For the `patronGold` map, you will use string keys to access patrons’ gold balance values. Try it out:

Listing 10.4 Accessing individual gold balances (Tavern.kt)

```
...
fun visitTavern() {
    ...
    while (patrons.size < 10) {
        patrons += "${firstNames.random()} ${lastNames.random()}"
    }

    println(patronGold)
    println(patronGold["Madrigal"])
    println(patronGold["Taarnyl"])
    println(patronGold["Eli"])

    narrate("$heroName sees several patrons in the tavern:")
    ...
}
...
```

Run NyetHack to print the balances for the three characters you specified:

```
...
{Taarnyl=86.0, Madrigal=4.5}
4.5
86.0
null
...
```

Note that the output includes only the values, not the keys. Also, your lookup for `"Eli"` returned `null`, because Eli has not been added to the `patronGold` map.

As with other collections, Kotlin provides functions for accessing the values stored in a map. [Table 10.1](#) shows some of the common map accessor functions and their behaviors.

Table 10.1 Map accessor functions

Function	Description	Example
<code>[]</code> (get/index operator)	Gets the value for a key; returns null if the key does not exist.	<code>patronGold["Reginald"]</code> <i>null</i>
getValue	Gets the value for a key; throws an exception if the key provided is not in the map.	<code>patronGold.getValue("Reggie")</code> <i>NoSuchElementException</i>
getOrDefault	Gets the value for the key or returns a default using a value you provide.	<code>patronGold.getOrDefault("Reginald", 0.0)</code> <i>0.0</i>
getOrElse	Gets the value for the key or computes a default using an anonymous function.	<code>patronGold.getOrElse("Reggie") {</code> <code>patronName -></code> <code>if (patronName == "Jane") 4.0</code> <code>else 0.0</code> <code>}</code> <i>0.0</i>

Adding Entries to a Map

Your map of patron gold values represents the purses of Madrigal and Taernyl, but it does not include purse values for the patrons you dynamically generate. Time to fix that by replacing `patronGold` with a **MutableMap**.

In your while loop that generates patrons, add an entry to the map to give each patron 6 gold when they enter the tavern. Also, remove the map entry lookups that you performed, since the keys are the patrons’ full names – not just their first names.

Listing 10.5 Populating the mutable map (Tavern.kt)

```
... visitTavern() {
    ...
    val patrons: MutableSet<String> = mutableSetOf()
    val patronGold = mapOf(
    val patronGold = mutableMapOf(
        TAVERN_MASTER to 86.00,
        heroName to 4.50
    )
    while (patrons.size < 10) {
        patrons += "${firstNames.random()} ${lastNames.random()}"
        val patronName = "${firstNames.random()} ${lastNames.random()}"
        patrons += patronName
        patronGold += patronName to 6.0
    }

    println(patronGold)
    println(patronGold["Madrigal"])
    println(patronGold["Taernyl"])
    println(patronGold["Eli"])

    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())
    ...
}
```

You used **Map**’s plus assign operator (`+=`) to add an entry to the map for each new key. (You can also use the **put** function, which works the same way.) If a mapping already exists for a key, the plus assign operator will overwrite the existing pair in the map. This was the same behavior you saw earlier when you attempted to create a map with duplicate keys.

[Table 10.2](#) shows some of the commonly used functions for modifying the contents of a mutable map.

Table 10.2 Mutable map mutator functions

Function	Description	Example
<code>=</code> (assignment operator)	Adds or updates the value in the map for the key specified.	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0) patronGold["Mordoc"] = 5.0 {Mordoc=5.0}</pre>
<code>+=</code> (plus assign operator)	Adds or updates one or more entries in the map based on the entry or map specified.	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0) patronGold += "Eli" to 5.0 {Mordoc=6.0, Eli=5.0} patronGold += mapOf("Eli" to 7.0, "Mordoc" to 1.0, "Sophie" to 4.5) {Mordoc=1.0, Eli=7.0, Sophie=4.5}</pre>
put	Adds or updates the value in the map for the key specified.	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0) patronGold.put("Mordoc", 5.0) {Mordoc=5.0}</pre>
putAll	Adds all the key-value pairs provided to the map.	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0) patronGold.putAll(listOf("Jebediah" to 5.0, "Sahara" to 6.0)) patronGold["Sahara"] 6.0</pre>
getOrPut	Adds an entry for the key if it does not exist already and returns the result; otherwise returns the existing entry.	<pre>val patronGold = mutableMapOf<String, Double>() patronGold.getOrPut("Randy") {5.0} 5.0 patronGold.getOrPut("Randy") {10.0} 5.0</pre>
remove	Removes an entry from the map and returns the value.	<pre>val patronGold = mutableMapOf("Mordoc" to 5.0) val mordocBalance = patronGold.remove("Mordoc") {} print(mordocBalance) 5.0</pre>
<code>-=</code> (minus assign)	Removes one or more entries from the map.	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0, "Jebediah" to 1.0, "Sophie" to 8.0, "Tariq" to 4.0</pre>

operator)		) patronGold -= listOf("Mordoc", "Sophie") {Jebediah=1.0, Tariq=4.0}
clear	Removes all entries from the map.	mutableMapOf("Mordoc" to 6.0, "Jebediah" to 1.0) .clear() {}

Modifying Map Values

Your **performPurchase** function does not currently charge patrons for their orders, which is an unsustainable business model for Taernyl. To perform a transaction, the price of the item should be deducted from the patron's purse and added to Taernyl's. The `patronGold` map associates gold balance values with a given patron's name as a key. You will modify the gold balance value for a patron to record the patron's new balance once the purchase is completed.

For now, Taernyl will use a fixed-pricing model, where every item costs the same amount. After you start charging your patrons, you will come back and use the prices in `tavern-menu-data.txt` instead.

Update **performPurchase** to use the `patronGold` map to track the flow of money through the tavern, making sure to decline orders that patrons cannot afford. Also, print the map before and after purchasing items to ensure that your patrons are being billed correctly.

Finally, having 10 patrons makes for a lively tavern, but it makes NyetHack's output really verbose. Adjust the number of patrons to a more manageable level.

Listing 10.6 Updating the values in `patronGold` (`Tavern.kt`)

```
fun visitTavern() {
    ...
    while (patrons.size < 10) {
    while (patrons.size < 5) {
        val patronName = "${firstNames.random()} ${lastNames.random()}"
        patrons += patronName
        patronGold += patronName to 6.0
    }
    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())

    println(patronGold)
    repeat(3) {
        placeOrder(patrons.random(), menuItems.random(), patronGold)
    }
    println(patronGold)
}

private fun placeOrder(patronName: String, menuItemName: String) {
private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
) {
    val itemPrice = 1.0

    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    if (itemPrice <= patronGold.getOrDefault(patronName, 0.0)) {
        narrate("$TAVERN_MASTER hands $patronName a $menuItemName")
        narrate("$patronName pays $TAVERN_MASTER $itemPrice gold")
        patronGold[patronName] = patronGold.getValue(patronName) - itemPrice
        patronGold[TAVERN_MASTER] = patronGold.getValue(TAVERN_MASTER) + itemPrice
    } else {
        narrate("$TAVERN_MASTER says, \"You need more coin for a $menuItemName\")
    }
}
```

Run NyetHack. You will see random orders and the characters' purse contents before and after the orders, along the lines of:

```
...
{Taernyl=86.0, Madrigal=4.5, Alex Fernsworth=6.0, Tariq Downstrider=6.0,...}
Alex Fernsworth speaks with Taernyl to place an order
Taernyl hands Alex Fernsworth a Shirley's Temple
Alex Fernsworth pays Taernyl 1.0 gold
Tariq Downstrider speaks with Taernyl to place an order
Taernyl hands Tariq Downstrider a Goblet of LaCroix
Tariq Downstrider pays Taernyl 1.0 gold
...
{Taernyl=89.0, Madrigal=4.5, Alex Fernsworth=5.0, Tariq Downstrider=5.0,...}
```

Converting Between Lists and Maps

Your next task in NyetHack is to charge your patrons the correct amount for their orders. To accomplish this, you can make another map that has names of menu items as keys and their prices as values. Previously, you solved the issue of parsing menu item names by introducing a new list called `menuItems` using the **List** constructor. Map does not have an equivalent constructor, but there are other ways to get your hands on a map.

Much like how you can convert a list to a set and vice versa, it is also possible to convert lists to maps with the **toMap** function – but with a slight catch. **toMap** is only available on lists that hold **Pairs**. So it is valid to call **toMap** on **List<Pair<String, Double>>**, but not on **List<String>**, for example. (If you attempt to call the **toMap** function on lists that hold anything other than **Pairs**, it will appear as if you are trying to call a function that has not been declared.)

To create a map for your menu item’s prices, you will start by making a list containing the key-value pairs for menu item names and prices. You will use the **List** constructor you saw in [Chapter 9](#) to dynamically populate this collection based on your menu item file. Because your list contains the key-value pairs of your menu, you can then call **toMap** to convert it to a map.

Listing 10.7 Converting a list to a map (Tavern.kt)

```
...
private val menuItems = List(menuData.size) { index ->
    val (_, name, _) = menuData[index].split(",")
    name
}

private val menuItemPrices: Map<String, Double> = List(menuData.size) { index ->
    val (_, name, price) = menuData[index].split(",")
    name to price.toDouble()
}.toMap()
...
```

This code works very similarly to how you initialized the `menuItems` list. It iterates over the menu and uses the same destructuring technique you saw before to read values on the menu. Unlike your `menuItems` initialization, you create pairs in the list and add a call to **toMap** after building the list. This allows you to dynamically read entries from your menu data file. Otherwise, to use **mapOf**, you would need code like this:

```
mapOf(
    "Dragon's Breath" to 5.91,
    "Shirley's Temple" to 4.12,
    "Goblet of LaCroix" to 1.22,
    "Pickled Camel Hump" to 7.33,
    "Iced Boilermaker" to 11.22
    "Hard Day's Work Ice Cream" to 3.21
    "Bite of Lembas Bread" to 0.59
)
```

With this price map in place, you can update **placeOrder** to apply the correct prices.

Listing 10.8 Charging full price for menu items (Tavern.kt)

```
...
private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
) {
    val itemPrice = 1.0
    val itemPrice = menuItemPrices.getValue(menuItemName)

    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    ...
}
```

Run NyetHack. Your output will now look like something like this:

```
...
{Taernyl=86.0, Madrigal=4.5, Mordoc Baggins=6.0, Sophie Ironfoot=6.0,...}
Mordoc Baggins speaks with Taernyl to place an order
Taernyl serves Mordoc Baggins a Goblet of LaCroix
Mordoc Baggins pays Taernyl 1.22 gold
Sophie Ironfoot speaks with Taernyl to place an order
Taernyl hands Sophie Ironfoot a Shirley's Temple
Sophie Ironfoot pays Taernyl 4.12 gold
...
{Taernyl=97.25, Madrigal=4.5, Mordoc Baggins=4.78, Sophie Ironfoot=1.88,...}
```

There is one more piece of data on the menu, which indicates what kind of food or drink an item is. You can use this information to generate slightly better wording in the output the player reads. Instead of “handing” food and drink to patrons, Taernyl might instead “serve” or “pour” the items. Create another map to store the menu items’ types, then use a `when` expression in **placeOrder** to tailor the output.

Listing 10.9 Parsing menu item types (Tavern.kt)

```
...
private val menuItemPrices: Map<String, Double> = List(menuData.size) { index ->
    val (_, name, price) = menuData[index].split(",")
    name to price.toDouble()
}.toMap()

private val menuItemTypes: Map<String, String> = List(menuData.size) { index ->
    val (type, name, _) = menuData[index].split(",")
    name to type
}.toMap()
...
private fun placeOrder(
    patronName: String,
```

```

menuItemName: String,
patronGold: MutableMap<String, Double>
) {
    val itemPrice = menuItemPrices.getValue(menuItemName)

    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    if (itemPrice <= patronGold.getOrDefault(patronName, 0.0)) {
narrate("$TAVERN_MASTER hands $patronName a $menuItemName")
        val action = when (menuItemTypes[menuItemName]) {
            "shandy", "elixir" -> "pours"
            "meal" -> "serves"
            else -> "hands"
        }
        narrate("$TAVERN_MASTER $action $patronName a $menuItemName")
        narrate("$patronName pays $TAVERN_MASTER $itemPrice gold")
        patronGold[patronName] = patronGold.getValue(patronName) - itemPrice
        patronGold[TAVERN_MASTER] = patronGold.getValue(TAVERN_MASTER) + itemPrice
    } else {
        narrate("$TAVERN_MASTER says, \"You need more coin for a $menuItemName\")
    }
}

```

Run NyetHack a few more times and confirm that your output uses the correct terminology to describe Taernyl’s actions.

```

...
Tariq Fernsworth speaks with Taernyl to place an order
Taernyl serves Tariq Fernsworth a Goblet of LaCroix
Tariq Fernsworth pays Taernyl 1.22 gold
Sophie Fernsworth speaks with Taernyl to place an order
Taernyl pours Sophie Fernsworth a Dragon's Breath
Sophie Fernsworth pays Taernyl 5.91 gold
Tariq Baggins speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Iced Boilermaker"
{Taernyl=87.22, Madrigal=4.5, Tariq Fernsworth=4.78, Tariq Baggins=6.0,
Sophie Baggins=6.0, Alex Fernsworth=6.0, Sophie Fernsworth=0.08999999999999986,
Sophie Ironfoot=6.0, Alex Baggins=6.0, Alex Downstrider=6.0,
Mordoc Fernsworth=6.0, Tariq Ironfoot=6.0}

```

Iterating Through a Map

You have updated the patrons’ gold balances, and only one task remains – formatting the results in a more suitable manner. You are currently showing these balances, but you likely want to use a less technical format that omits any brackets or equals signs and that rounds balances for patrons like Sophie Fernsworth, whose balance above is being reported as 0.089999999999999986 due to floating point precision loss. You will improve your output formatting by iterating through your map with **forEach**.

Add a new function called **displayPatronBalances** at the end of `Tavern.kt` that iterates through the `patronGold` map, printing the final gold balance (formatted to the second decimal place, as you did in [Chapter 5](#)) for each patron. Call it at the end of the **visitTavern** function.

Listing 10.10 Displaying patron balances (Tavern.kt)

```
...
fun visitTavern() {
    ...
    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())

    println(playerGold)
    repeat(3) {
        placeOrder(patrons.random(), menuItems.random(), patronGold)
    }
    println(playerGold)
    displayPatronBalances(patronGold)
}
...
private fun displayPatronBalances(patronGold: Map<String, Double>) {
    narrate("$heroName intuitively knows how much money each patron has")
    patronGold.forEach { (patron, balance) ->
        narrate("$patron has ${"%0.2f".format(balance)} gold")
    }
}
```

This **forEach** call is very similar to the one you used with lists and sets in [Chapter 9](#), but notice two details in your newly added code: First, your lambda expression argument uses destructuring. This call to **forEach** takes in a lambda expression of type **(Pair<String, Double>) -> Unit**. Typically, you would use `it.first` and `it.second` to access the key and value of the map entry, but here you take advantage of destructuring in the lambda expression’s parameter list to add more clarity to your code.

Second, you declared the parameter to **displayPatronBalances** as a **Map**, instead of a **MutableMap**. Why? You do not need – nor do you want – to edit the patrons’ gold balances when displaying them.

When your function accepts a collection but does not need to make changes to it, it is good practice to prefer the read-only collection type instead of the mutable one. For one thing, it prevents you from accidentally modifying the map when you only intend to read from it. As an added bonus, this practice lets other parts of your program call your function regardless of whether they use **Map** or **MutableMap**.

But this raises a question: Why does Kotlin allow you to pass a **MutableMap** as an argument of the **Map** type? After all, **MutableMap** and **Map** are different types.

Kotlin lets you *cast* **MutableMaps** to the **Map** type. Casting lets your program interpret values as a different type than the one they were defined or declared as. **MutableMap** builds on the functionality of **Map**, so you are permitted to treat your map as a more general type (**Map**) than it actually is (**MutableMap**). In this particular scenario, the cast is implicit and happens when you call **displayPatronBalances**.

Keep in mind that because casting is a reinterpretation of an existing value, changes from elsewhere in your code to the original map (which, remember, is still a **MutableMap**) will affect everything that holds a reference to the map, regardless of how it has been cast. You can prevent these updates from appearing by calling **toMap** on a **MutableMap** to make a read-only copy instead of casting the map.

The same casting trick works with **MutableList** and **MutableSet**, making it easy to treat any collection as its read-only variation. But you cannot go in the opposite direction, because it is not true that every **Map** is a **MutableMap**.

If you want to safely convert a read-only **Map** to a **MutableMap**, you can use **toMutableMap**. Bear in mind that you will receive a copy of the original map, and insertions or deletions will not affect the original read-only copy. (If you want to perform this dangerous casting operation, you will see techniques to perform manual casting in [Chapter 15](#) and [Chapter 17](#).)

Run NyetHack, sit back, and watch as the patrons of Taernyl’s Folly chat with the tavern master, order off the menu, and pay for their items:

```
A hero enters the town of Kronstadt. What is their name?
Madrigal
Madrigal, The Renowned Hero, heads to the town square
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
```

Madrigal sees several patrons in the tavern:
Tariq Ironfoot, Alex Baggins, Alex Fernsworth, Sophie Ironfoot, Tariq Downstrider
Alex Fernsworth speaks with Taernyl to place an order
Taernyl pours Alex Fernsworth a Shirley's Temple
Alex Fernsworth pays Taernyl 4.12 gold
Alex Fernsworth speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Shirley's Temple"
Tariq Ironfoot speaks with Taernyl to place an order
Taernyl pours Tariq Ironfoot a Dragon's Breath
Tariq Ironfoot pays Taernyl 5.91 gold
Madrigal intuitively knows how much money each patron has
Taernyl has 96.03 gold
Madrigal has 4.50 gold
Tariq Ironfoot has 0.09 gold
Alex Baggins has 6.00 gold
Alex Fernsworth has 1.88 gold
Sophie Ironfoot has 6.00 gold
Tariq Downstrider has 6.00 gold

In the last two chapters, you learned how to work with Kotlin’s **List**, **Set**, and **Map** collection types. [Table 10.3](#) compares their features.

Table 10.3 Kotlin collections summary

Collection type	Ordered?	Unique?	Stores	Supports destructuring?
List	Yes	No	Elements	Yes
Set	No	Yes	Elements	No
Map	No	Keys	Key-value pairs	No

Since collections are read-only by default, you must explicitly create a mutable collection (or convert a read-only collection to be mutable) to modify its contents – preventing you from accidentally adding or removing elements.

In the next chapter, you will learn more advanced functional programming techniques that let you work even more effectively with Kotlin’s collection types.

Challenge: Complex Orders

Currently, a patron at Taernyl's Folly must order one item at a time. If they want to order both food and drink, they must place two separate orders. Flesh out the tavern's ordering system by allowing patrons to place orders for multiple items.

Patrons should place orders with 1-3 items in their order, with both the number and the items selected randomly. If a patron cannot afford their entire order, the sale should be denied, and they should not be served any items until they place a different order. You may need to give patrons more gold to afford some more expensive meal combinations.

(Be sure to save a copy of NyetHack before you make these changes. The next chapter will introduce many changes to the way you use collections that are incompatible with the changes you will make in this challenge.)

Functional Programming Basics

There are many paradigms that a programming language can enable developers to use. Two of the most commonly used and well-known paradigms that Kotlin implements are object-oriented programming and functional programming. You will learn about object-oriented programming in [Part IV](#) of this book; this chapter focuses on functional programming.

Kotlin supports multiple programming styles, so you can mix object-oriented and functional programming styles to suit the problem at hand. Some programming languages (like Haskell) *only* support functional programming. These purely functional languages are more commonly used in academia than commercial software, but the concepts and many of the techniques have been adopted by other languages like JavaScript and Swift.

In [Chapter 8](#), you learned about functions that accept another function as a parameter, functions that return a function as their result, and function types, which enable you to define functions as values. Functions that accept a function as a parameter or return a function are called [higher-order functions](#). [Functional programming](#) relies on data that is returned from a small number of higher-order functions designed specifically to work on collections. Functions in functional programming are designed to be [composable](#), meaning that simple functions can be combined to build complex behavior.

This chapter explores some of the functional programming features Kotlin offers and explains the ideas behind the functional programming paradigm. Although this discussion is Kotlin-specific, many of the ideas – and even some of the function names – also appear in other programming languages.

You will be using functional programming to update `Tavern.kt`. More specifically, you will be using functional programming with your collections, which will make the Tavern's codebase both more capable and more concise. We will group these functional programming concepts into three broad concepts: *transforming* data, *filtering* data, and *combining* data.

Transforming Data

The first category we will discuss is transformation functions. A [transform function](#) modifies each element of a collection by walking through its contents and modifying each item with a [transformer function](#) provided as an argument. The transform function then returns the modified collection.

Two commonly used transform functions are **map** and **flatMap**.

map

The **map** transform function iterates through the collection it is called on and applies its transformer function to each element. The result is a collection of the transformed elements. (Keep in mind that we are talking about the **map** function, not to be confused with the **mapOf** function or the **Map** type.)

Both the **map** and **flatMap** functions are commonly used to convert a given data set into a different representation of the same values. Take another look at the code you used to parse the menu items out of `tavern-menu-data.txt`:

```
private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")

private val menuItems = List(menuData.size) { index ->
    val (_, name, _) = menuData[index].split(",")
    name
}
```

This code works, but the computation of `menuItems` is a bit opaque. You make a new list with the same number of items as `menuData`, and then populate the new list by iterating over `menuData`. It is very common to use **map** to achieve this instead.

Update your `menuItems` declaration to use **map**, as shown in [Listing 11.1](#). We will explain how the **map** function works after you make these changes.

Listing 11.1 Using the map function (Tavern.kt)

```
...
private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")

private val menuItems = List(menuData.size) { index ->
    val (_, name, _) = menuData[index].split(",")
    name
}
private val menuItems: List<String> = menuData.map { menuEntry: String ->
    val (_, name, _) = menuEntry.split(",")
    name
}
...
```

Run `NyetHack` and confirm that your output has not changed:

```
...
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
...
```

As before, `menuItems` is assigned to a list containing the names of the items available at Taernyl's Folly. But now, when your program runs, **map** creates the list to hold elements returned by its transformer function. It then runs the transformer function on each element (in this case, reading the second element from each value in `menuData`), and appends each transformed value to the list, preserving the order of the original collection. Finally, **map** returns the new list of mapped values.

Take another look at the types you used. `menuData` starts out as a **List<String>**, and the lambda expression you pass to **map** returns a **String**. Here, you are mapping to values of the same type, which is certainly allowed and happens very commonly when you use a transformation function.

However, the **map** function lets you arbitrarily map to any type. For example, you might have a list of numbers encoded as strings. If you wanted to convert all the strings into doubles, you could use **map** in conjunction with the **toDouble()** function:

```
val numbers: List<String> = listOf("1.0", "2.0", "3.0")
["1.0", "2.0", "3.0"]

val numbersAsDoubles: List<Double> = numbers.map { it.toDouble() }
[1.0, 2.0, 3.0]
```

For your first **map** call, you provided a named parameter for the transformation, including its type information: `menuEntry: String`. We had you do this for clarity, but neither the name nor the type declaration is required. You could also have used the `it` identifier, and you can drop the type information entirely since Kotlin is able to infer it. To see what this call would more likely look like in the real world, remove the explicit type information – but keep

the name, since it is providing helpful context.

Listing 11.2 Inferring map’s types (Tavern.kt)

```
...
private val menuItems: List<String> = menuData.map { menuEntry: String ->
    val (_, name, _) = menuEntry.split(",")
    name
}
...
```

This combination of expressiveness and conciseness is invaluable for applications of all sizes, and you will see **map** used in many codebases.

Update the definitions of `menuItemPrices` and `menuItemTypes` to also use the **map** function:

Listing 11.3 Using map to make Maps (Tavern.kt)

```
...
private val menuItems = menuData.map { menuEntry ->
    val (_, name, _) = menuEntry.split(",")
    name
}

private val menuItemPrices: Map<String, Double> = List(menuData.size) { index ->
    val (_, name, price) = menuData[index].split(",")
}
private val menuItemPrices = menuData.map { menuEntry ->
    val (_, name, price) = menuEntry.split(",")
    name to price.toDouble()
}.toMap()

private val menuItemTypes: Map<String, String> = List(menuData.size) { index ->
    val (type, name, _) = menuData[index].split(",")
}
private val menuItemTypes = menuData.map { menuEntry ->
    val (type, name, _) = menuEntry.split(",")
    name to type
}.toMap()
...
```

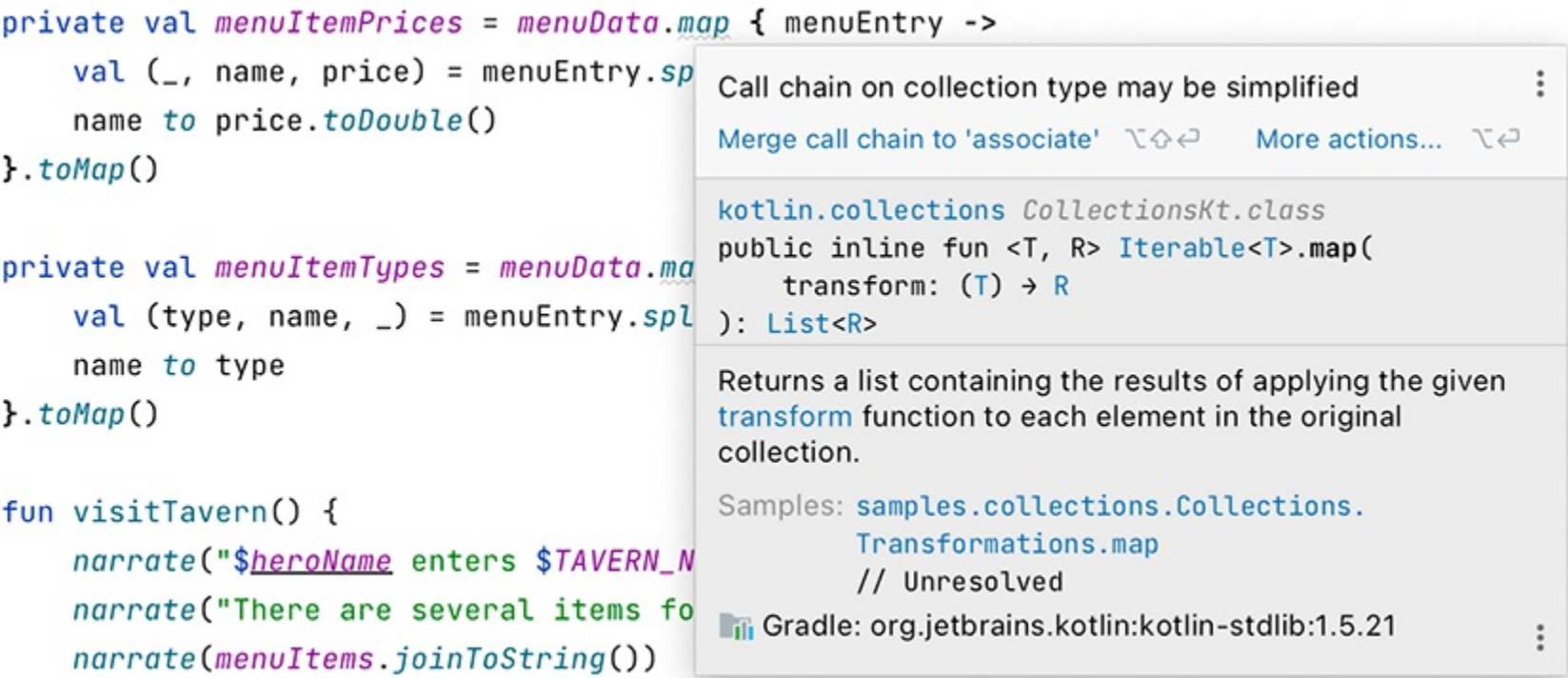
Despite its name, **map** is returning a **List** of **Pairs** for both `menuItemPrices` and `menuItemTypes`. This is why you still need your **toMap** calls.

Run `NyetHack` to confirm that nothing has changed in the output.

associate

When you made the changes above in your code, you might have noticed that IntelliJ added faint squiggly lines under the calls to **map**. This is IntelliJ’s way of subtly beckoning for your attention. Hover over either **map** call with your mouse to see what advice IntelliJ has to offer ([Figure 11.1](#)).

Figure 11.1 Functional programming hints



Here, IntelliJ is pointing you toward a different function called **associate**. This one function call acts the same as calling `yourCollection.map { key to value }.toMap()`. Try it out, either by clicking Merge call chain to ‘associate’ in the pop-ups or by making the changes manually:

Listing 11.4 Using associate (Tavern.kt)

```
...
private val menuItemPrices = menuData.map { menuEntry ->
    val (_, name, price) = menuEntry.split(",")
    name to price.toDouble()
}.toMap()

private val menuItemTypes = menuData.map { menuEntry ->
    val (type, name, _) = menuEntry.split(",")
    name to type
}.toMap()
```

...

Kotlin has many functional programming operations like this up its sleeve – too many to show in this book. We have opted to show the most fundamental operations in this chapter. When combined, you can achieve very complex operations by composing functions.

Sometimes you will accidentally write a chain of operations that is already implemented by the standard library, as you did here. IntelliJ can often detect this and give you hints to use the built-in version. This can be a great way of learning new functional programming operations when working with collections in your code. We encourage you to keep your eyes peeled for recommendations like this from IntelliJ.

Destructuring with functional programming

`menuItems`, `menuItemPrices`, and `menuItemTypes` have a fair amount of duplication among them – they each split the elements from `menuData` and pull out specific components. Because `menuData`’s lines are always split, you can move the `split(",")` call to `menuData` itself using **map**. When you do this, you will also be able to use the destructuring syntax with lambda expressions that you saw in the last chapter.

Make this change, which will make your lambda expressions even more concise and expressive.

Listing 11.5 Using destructuring with map (Tavern.kt)

```
...
private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")
    .map { it.split(",") }

private val menuItems = menuData.map { menuEntry ->
    val (_, name, _) = menuEntry.split(",")
    name
}
private val menuItems = menuData.map { (_, name, _) -> name }

private val menuItemPrices = menuData.associate { menuEntry ->
    val (_, name, price) = menuEntry.split(",")
    price.toDouble()
}
private val menuItemPrices = menuData.associate { (_, name, price) ->
    name to price.toDouble()
}

private val menuItemTypes = menuData.associate { menuEntry ->
    val (type, name, _) = menuEntry.split(",")
    type
}
private val menuItemTypes = menuData.associate { (type, name, _) ->
    name to type
}
...
```

You place your new call to **split** inside a **map** call, because you want to split each of the lines after they have been separated by the previous **split**, which returns a list. If you tried to `split(",")` directly on the result of `split("\n")`, you would be greeted with a compiler error.

It is very common in Kotlin to see functional operations like this on just a few lines – or even one line – of code. Compare your newly updated code to what you had before. Although your new code requires knowledge of the **map** and **associate** functions, the intent is clearer. Compared to other approaches, Kotlin’s functional programming APIs often lead to code that is more transparent about its intent and is much more concise.

Run NyetHack once again to ensure you have no errors.

flatMap

The next transform function we will introduce is called **flatMap**. **flatMap** behaves nearly identically to **map** except that it *flattens* the mapped values: It takes a collection of collections (say, a list of lists of strings) and returns a collection of the nested type (like a list of strings).

Right now, you do not have a need for **flatMap**, but Taernyl is about to change that. The tavern master would like to mark something on the menu as the item of the day to promote it. To figure out which item will be chosen, he wants to choose from among his patrons’ favorite items.

For many of Taernyl’s patrons, picking a favorite item is easy. But others might have several favorite items. Alex Ironfoot, for example, has a sweet tooth and craves all the desserts on the menu. Create a new function to determine a patron’s favorite menu items.

Listing 11.6 Asking for patrons’ favorites (Tavern.kt)

```
import java.io.File
import kotlin.random.Random
import kotlin.random.nextInt

...
fun visitTavern() {
    ...
}

private fun getFavoriteMenuItems(patron: String): List<String> {
    return when (patron) {
        "Alex Ironfoot" -> menuItems.filter { menuItem ->
            menuItemTypes[menuItem]?.contains("dessert") == true
        }
        else -> menuItems.shuffled().take(Random.nextInt(1..2))
    }
}
```

```

}
private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
) {
    ...
}
...

```

This new code takes advantage of three functions on **List**: **filter**, **shuffled**, and **take**. The first condition, which handles Alex Ironfoot, uses **filter** to find all desserts on the menu. We will talk about this function more in [the section called *Filtering Data*](#).

The **shuffled** function returns a copy of the original list with its elements randomly reordered. The **take** function returns a list with up to a given number of items – in this case, either one or two items at random. Putting the two together, patrons other than Alex Ironfoot will randomly pick one or two items from the menu to be their favorites.

When picking the item of the day, Taernyl should consider who is in the tavern to find out all patrons’ favorite items. Obtain this data by mapping patrons to their favorite menu items.

Listing 11.7 Using map to get favorites (Tavern.kt)

```

...
fun visitTavern() {
    ...
    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())

    val favoriteItems = patrons.map { getFavoriteMenuItems(it) }
    println("Favorite items: $favoriteItems")
    ...
}
...

```

What do you think this code will print? Run NyetHack to see for yourself.

```

...
Favorite items: [[Pickled Camel Hump], [Dragon's Breath, Goblet of LaCroix],
    [Iced Boilermaker, Shirley's Temple], [Pickled Camel Hump]]
...

```

Is this what you expected?

Take a closer look at the brackets in this output: The items in this output are grouped into lists. Now, highlight the **favoriteItems** variable and press Control-Shift-P to show its type. IntelliJ will show a tooltip indicating that it is a **List<List<String>>**.

To understand why **favoriteItems** is a list of lists, recall how the **map** function works. The return values from the lambda expression you provide to **map** are used directly in the resulting list. Here, the transformation function returns a list – so you end up with a list of lists.

Sometimes you want this nesting of lists (**menuData** is also a **List<List<String>>**, which works fine), but other times nesting like this gets in the way and makes it difficult to use your collections. To remove the nesting, you *flatten* the collection.

There are two ways to go about flattening a nested collection like this. One option is to use the **flatten** function, which will remove the nesting. But if you are already using the **map** transformation – as you are here – you can map and flatten in a single step using **flatMap**. Try it out by replacing the call to **map** when figuring out your patrons’ favorite items.

Listing 11.8 Using flatMap to get favorites (Tavern.kt)

```

...
fun visitTavern() {
    ...
    val favoriteItems = patrons.map { getFavoriteMenuItems(it) }
    val favoriteItems = patrons.flatMap { getFavoriteMenuItems(it) }
    println("Favorite items: $favoriteItems")
    ...
}
...

```

Run NyetHack again. You will now see your patrons’ favorites printed without the nested lists:

```

...
Favorite items: [Pickled Camel Hump, Iced Boilermaker, Goblet of LaCroix,
    Iced Boilermaker, Iced Boilermaker, Pickled Camel Hump]
...

```

Notice how easy it is to modify operations defined using functional programming. Here, you removed an entire level of nesting from your output simply by changing **map** to **flatMap**. Without these functional programming techniques, you might find yourself rewriting algorithms (or appending new algorithms onto existing ones) to change your output in ways that could be accomplished with just a few keystrokes in the functional world.

Now that Taernyl has a list of favorite items (and not a list of lists of favorite items), he is equipped to determine the item of the day. Use the **random** function to begin Taernyl’s new marketing campaign.

Listing 11.9 Picking an item of the day (Tavern.kt)

```

...
fun visitTavern() {

```

```

...
val favoriteItems = patrons.flatMap { getFavoriteMenuItems(it) }
val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()
println("Favorite items: $favoriteItems")
narrate("The item of the day is the $itemOfDay")
...
}
...

```

Run NyetHack. Your output should now look like this:

```

...
The item of the day is the Goblet of LaCroix
...

```

map vs flatMap

map and **flatMap** are extremely similar functions. You may be wondering when you should use one or the other.

When deciding between **map** and **flatMap**, ask yourself “Does my transformation function return a collection?” If the answer is no, then use **map**. If your transformation function does not return a **List** or other collection, you cannot use **flatMap**, because it requires that your lambda expression return a collection of elements.

If your transformation function *does* return a collection of elements – and not just a single element – then you *might* want to use **flatMap**, depending on the type you want to end up with. If you want a collection of collections, then stick to **map**. If you want to remove the inner nesting, use **flatMap**.

Filtering Data

The second category of functions in functional programming is filters. A [filter function](#) accepts a function that determines which elements should appear in the returned list. There are several ways filtering functions can take shape.

One example is the **take** function you saw earlier, which discards all items after a target number of elements has been reached. There is also a related function called **drop** that discards some number of elements from the start of a collection.

But the most general filtering function is aptly named **filter**.

filter

The **filter** function accepts a predicate function that checks each element in a collection against a condition and returns a Boolean value. If the predicate returns true, the element is added to the new collection the filter returns. If the predicate returns false, the element is excluded from the new collection.

Taernyl has a fine establishment and wants to maintain the elite status of his tavern. He wants to enact a rule that patrons who are running out of gold must leave the tavern. Remove patrons from Taernyl’s Folly if they have less than four gold, using **filter** to detect the offending patrons.

Listing 11.10 Finding departing patrons with filter (Tavern.kt)

```
...
fun visitTavern() {
    ...
    displayPatronBalances(patronGold)

    val departingPatrons: List<String> = patrons
        .filter { patron -> patronGold.getOrDefault(patron, 0.0) < 4.0 }
    departingPatrons.forEach { patron ->
        narrate("$heroName sees $patron departing the tavern")
    }

    narrate("There are still some patrons in the tavern")
    narrate(patrons.joinToString())
}
```

Run NyetHack and pay particular attention to the last part of your output:

```
...
Madrigal sees Tariq Baggins departing the tavern
There are still some patrons in the tavern
Sophie Fernsworth, Sophie Downstrider, Mordoc Downstrider, Tariq Baggins,
    Alex Downstrider
```

(You might need to run a few times until a patron purchases a sufficiently expensive item.)

In the sample output above, notice that while Madrigal sees patrons leaving the tavern, they are still included in the `patrons` list. This is because **filter** – like many operations, including **map** – returns a *new* list. The original collection remains unmodified, regardless of whether it is mutable.

To make the patrons actually depart, you also need to remove them from both `patrons` and `patronGold`.

Listing 11.11 Removing departed patrons (Tavern.kt)

```
...
fun visitTavern() {
    ...
    val departingPatrons: List<String> = patrons
        .filter { patron -> patronGold.getOrDefault(patron, 0.0) < 4.0 }
    patrons -= departingPatrons
    patronGold -= departingPatrons
    departingPatrons.forEach { patron ->
        narrate("$heroName sees $patron departing the tavern")
    }

    narrate("There are still some patrons in the tavern")
    narrate(patrons.joinToString())
}
...
```

Run NyetHack again to confirm that your patrons are in fact leaving the tavern.

```
...
Madrigal sees Tariq Downstrider departing the tavern
There are still some patrons in the tavern
Alex Ironfoot, Mordoc Fernsworth, Sophie Baggins
```

Combining Data

The third category of functions used in functional programming is [combines](#). Combining functions take multiple collections and merge them into a new one. (This is different than **flatMap**, which is called on one collection that contains other collections.)

zip

One example of a combining function is **zip**. When you **zip** two lists together, you combine the items from the two lists in the order they appear. For example, if you zipped the lists [1, 2, 3] and ["a", "b", "c"], you could create a result like ["1a", "2b", "3c"].

One application of the **zip** function would be to generate patron names. Instead of randomly picking first and last name pairs until you generate five unique names, you can use zip to ensure that all your patrons have unique, randomly paired first and last names. Try it out:

Listing 11.12 Zipping names together (Tavern.kt)

```
... visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    narrate(menuItems.joinToString())

    val patrons: MutableSet<String> = mutableSetOf()
    val patrons: MutableSet<String> = firstNames.shuffled()
        .zip(lastNames.shuffled()) { firstName, lastName -> "$firstName $lastName" }
        .toMutableSet()

    val patronGold = mutableMapOf(
        TAVERN_MASTER to 86.00,
        heroName to 4.50
    )
    while (patrons.size < 5) {
        val patronName = "${firstNames.random()} ${lastNames.random()}"
        patrons += patronName
    patrons.forEach { patronName ->
        patronGold += patronName to 6.0
    }

    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())
    ...
}
```

The lambda expression you provide tells **zip** how to combine the elements from the two collections. Here, you are appending the first name and a space to the last name to form a full name. You can also omit this transformer function entirely, which would cause **zip** to return a list of pairs instead. (It would be the same as if you provided the argument { firstName, lastName -> firstName to lastName }).

Run your new code. Your patron names will again randomly generate, but this time their first names and last names will be unique. For example, your patrons might now be Sophie Downstrider, Alex Fernsworth, Tariq Baggins, and Mordoc Ironfoot.

Your `firstNames` and `lastNames` sets have four elements each, so **zip** returns a list with four elements. What if the inputs were of different sizes? **zip** returns a list that is the same size as the smaller of the two input collections. Once every element in the smaller input has been zipped to a mate, the remaining elements in the larger input collection are silently ignored.

Since you are no longer adding an entry to `patronGold` when creating each patron individually, you used **forEach** to loop over each patron and assign their gold. This works, but you can populate the entire map with your single **mutableMapOf** call. Remove the **forEach** call as shown in [Listing 11.13](#), making sure to add the comma after `heroName to 4.50`. We will explain the new syntax after you enter it.

Listing 11.13 Spreading arguments (Tavern.kt)

```
... visitTavern() {
    ...
    val patrons: MutableSet<String> = firstNames.shuffled()
        .zip(lastNames.shuffled()) { firstName, lastName -> "$firstName $lastName" }
        .toMutableSet()

    val patronGold = mutableMapOf(
        TAVERN_MASTER to 86.00,
        heroName to 4.50,
        *patrons.map { it to 6.00 }.toTypedArray()
    )
    patrons.forEach { patronName ->
        patronGold += patronName to 6.0
}

    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())
    ...
}
```

mutableMapOf takes a variable number of arguments, which you can learn more about in [the section called For](#)

[*the More Curious: The vararg Keyword*](#) near the end of this chapter. Normally, you cannot pass a **List** into any of the collection builders – unless you want a collection of lists. Instead, you need to spread the values of the collection into separate arguments. You did this here with the [*spread operator*](#) (*).

The spread operator causes the elements of a collection to be treated as individual parameters to functions that accept a variable number of arguments. One limitation of the spread operator is that it only works on **Arrays**, hence the need to also call **toTypedArray**. Although the spread operator has a very tight niche, it is great when you need to build collections in this way.

Why Functional Programming?

Look back at the way **zip** is used to create a list of patron names in [Listing 11.12](#). Imagine implementing the same task without these functional programming APIs. In Java, for example, it might look something like this:

```
List<String> firstNames = Arrays.asList("Alex", "Mordoc", "Sophie", "Tariq");
List<String> lastNames = Arrays.asList("Ironfoot", "Fernsworth", "Baggins",
    "Downstrider");

Collections.shuffle(firstNames);
Collections.shuffle(lastNames);

List<String> patrons = new ArrayList<>();
for (int i = 0; i < firstNames.size; i++) {
    patrons.add(firstNames.get(i) + " " + lastNames.get(i));
}
```

This style of programming is called [*imperative programming*](#). At first glance, the imperative version here may look like it accomplishes the task in roughly the same number of lines as the functional version in [Listing 11.12](#). But the functional approach offers some key benefits, including the perk of implicit [*accumulator variables*](#).

Accumulator variables (like `patrons`, in this example) are defined implicitly in functional programming operations, eliminating the need for temporary variables that hold intermediate calculations. The results from functional operations are added to accumulators automatically, reducing the risk of bugs.

This is what makes it so easy to add new operations to a functional chain. Compare that to operations in the imperative style: Without implicit accumulator variables, new operations usually involve creating a new temporary variable to assist with your transformations.

Another reason it is so easy to add steps to functional chains is that all the functional operations are designed to work with iterables. Suppose the `patrons` map needed to be formatted to represent orders after building the map. In the imperative style, that would require an addition like this:

```
List<String> formattedOrders = new ArrayList<>();
for (Map.Entry<String, String> favoriteOrder : customerFavorites.entrySet()) {
    formattedOrders.add(favoriteOrder.getKey() + " orders their favorite item - "
        + favoriteOrder.getValue());
}
```

A new accumulator value and a new `for` loop that works to populate the accumulator with results – more entities, more state, more to keep track of.

With the functional style, subsequent operations are easily added to the chain without the need for additional state. The same program could be implemented functionally with the simple addition of:

```
.map { "${it.key} orders their favorite menu item - ${it.value}" }
```

Sequences

In [Chapter 9](#) and [Chapter 10](#), you were introduced to the collection types **List**, **Set**, and **Map**. These collection types are all known as *eager collections*. When an instance of any of these types is created, all the values it contains are added to the collection and can be accessed.

There is another flavor of collection: *lazy collections*. The term *lazy* indicates that a value is not created until the first time it is requested. Lazy collection types can provide better performance – especially when working with very large collections – because their values are produced only as needed.

Kotlin offers a built-in lazy collection type called **Sequence**. **Sequences** do not index their contents, and they do not keep track of their size. In fact, when working with a sequence, the possibility of an infinite sequence of values exists, because there is no limit to the number of items that can be produced.

With a sequence, you define a function that is referred to each time a new value is requested, called an *iterator function*. One way to define a sequence and its iterator is by using a sequence builder function provided by Kotlin, **generateSequence**. This function accepts an initial seed value, the starting place for the sequence. When the sequence is acted on by another function, **generateSequence** calls an iterator you specify that determines the next value to produce. For example:

```
generateSequence(0) { it + 1 }
    .onEach { println("The Count says: $it, ah ah ah!") }
```

If you were to run this snippet, the **onEach** function would execute forever.

So, what is a lazy collection good for, and why choose it over a list? Suppose you wanted to write a block of code that finds the first N primes – say, 1,000. A first shot at an implementation might look like this:

```
// Determines whether a number is prime
fun isPrime(number: Int): Boolean {
    (2 until number)
        .map { divisor ->
            if (number % divisor == 0) {
                return false // Not a prime
            }
        }
    return true
}

val listOfPrimes = (1..5000)
    .toList()
    .filter { isPrime(it) }
    .take(1000)
```

The problem with this implementation is that you do not know how many numbers you have to check to get 1,000 primes. This implementation takes a guess – 5,000 – but in fact this is not enough. (It will only get you 669 primes, if you want to know.)

This is a perfect case for using a lazy collection, instead of an eager one, to back the chain of functions. A lazy collection is ideal, because you do not need to define an upper bound for the number of items to check for the sequence:

```
val oneThousandPrimes = generateSequence(3) { value ->
    value + 1
}.filter { isPrime(it) }
    .take(1000)
```

In this solution, **generateSequence** produces a new value, one at a time, starting from 3 (the seed value) and incrementing by one each time. Then it filters the values with the extension **isPrime**. It continues doing this until 1,000 items have been produced. Because there is no way to know how many candidate numbers will have to be checked, lazily producing new values until the **take** function is satisfied is ideal.

In most cases, the collections you work with will be small, containing fewer than 1,000 elements. In these cases, worrying about using a sequence or a list for a constrained number of items will be of little concern, because the performance difference between the two collection types will be negligible – on the order of a few nanoseconds.

But with more sizable collections, with hundreds of thousands of elements, the performance improvement to be realized by switching the collection type can be significant. In these cases, you can convert a list to a sequence quite simply:

```
val listOfNumbers = (0 until 100000000).toList()
val sequenceOfNumbers = listOfNumbers.asSequence()
```

The functional programming paradigm can require frequent creation of new collections, and sequences provide a scalable mechanism for working with large collections.

In this chapter, you saw how to use basic functional programming tools like **map**, **filter**, and **zip** to streamline how you work with data. You also saw how to use sequences to work efficiently as your data set grows larger.

In the next chapter, you will wrap up your tour of functional programming concepts by learning about Kotlin's scope functions.

For the More Curious: Profiling

When the speed of code is an important consideration, Kotlin provides utility functions for profiling code performance: **measureNanoTime** and **measureTimeInMillis**. Both functions accept a lambda as their argument and measure the execution speed of the code contained within the lambda. **measureNanoTime** returns a time in nanoseconds, and **measureTimeInMillis** returns a time in milliseconds.

Wrap the function to measure in one of the utility functions like so:

```
val listInNanos = measureNanoTime {  
    // List functional chain here  
}  
  
val sequenceInNanos = measureNanoTime {  
    // Sequence functional chain here  
}  
  
println("List completed in $listInNanos ns")  
println("Sequence completed in $sequenceInNanos ns")
```

As an experiment, try profiling the performance of the list and sequence versions of the prime number examples in the REPL. (Change the list example to check numbers through 7,919 so it can find 1,000 primes.) How much does the change from a list to a sequence affect the performance time?

For the More Curious: Aggregating Data

We have shown a small set of the functional programming APIs that exist in Kotlin. There is another group of functions called [aggregation](#) functions that can reduce the entire contents of a collection into a single value. The best known of these functions is called **reduce**, and there is a similar function called **fold**.

reduce

reduce accumulates all values in a given collection into a single output value. The **reduce** function accepts a single lambda expression. This lambda expression takes in two parameters: A running aggregated value, called the *accumulator*, and the next value in the collection to aggregate. The return value from this lambda expression will be used as the next value for the accumulator.

There are a number of scenarios where you might reach for **reduce**. For example, consider a table of patrons: Each patron might place their own order, but the kitchen might treat this group of orders as a single order. To perform this transformation, you could **reduce** the patrons' orders into a single order for the table. The corresponding code might look like this:

```
val ordersAtTable: List<Order> = listOf(...)
val tableOrder: Order = ordersAtTable.reduce { acc, order -> acc + order }
```

(This example uses an **Order** type. You do not have such a type, but you could create it yourself. You will learn how to define your own types in [Part IV](#) of this book.)

Note that **reduce** returns a single element instead of a collection. This value will be the final accumulated result. If the list only has one element in it, your lambda expression will not be called, and the first (and only) value in the collection will be returned.

fold

Another function that is useful for aggregating values is **fold**. **fold** behaves similarly to **reduce**, with some important distinctions.

The **fold** function requires an initial accumulator value, instead of using the first value in the collection. As with **reduce**, this accumulator value is updated with the result of a lambda that is called for each item. One big advantage that **fold** has over **reduce** is that you can make the accumulator any type you desire (**reduce**'s accumulator must be the type contained in the list).

Suppose that patrons in Taernyl's Folly are responsible for paying sales tax and adding a gratuity when paying for their orders. (Taernyl's patrons are generous and always tip 20% after sales tax for his excellent service.) You could use **fold** to determine the price of an order:

```
val orderSubtotal = menuItemPrices.getDefault("Dragon's Breath", 0.0)

val salesTaxPercent = 5
val gratuityPercent = 20
val feePercentages: List<Int> = listOf(salesTaxPercent, gratuityPercent)

val orderTotal: Double = feePercentages.fold(orderSubtotal) { acc, percent ->
    acc * (1 + percent / 100.0)
}

println("Order subtotal: $orderSubtotal")
println("Order total: $orderTotal")
```

The initial value for the accumulator, `orderSubtotal` (5.91 in the case of Dragon's Breath), is passed to the lambda expression as `acc`, and the first fee percentage – sales tax (5%) – is passed in as `percent`. (The default name of this argument is `item`.) The lambda expression then converts the integer percent to a multiplier, computes the product to determine the price with the additional charge, and returns the updated accumulated value.

In the next calculation, the new accumulated value – 6.2055 for Dragon's Breath – is passed to the lambda expression as `acc`, and the second fee percentage – the gratuity (20%) – is passed in as `percent`. The final accumulator value holds the result (7.4466 for Dragon's Breath) and is returned by **fold**.

sumBy

If you have a collection with values that you want to add together, you can also use **sumBy** (for adding **Ints**) or **sumByDouble** (for floating point values). For example, if you wanted to compute the price for all items on the menu, you could perform this calculation using **sumByDouble** like this:

```
val orderTotal = menuItems.sumByDouble { item ->
    menuItemPrices.getDefault(item, 0.0)
}
```

```
}  
println("Order price: $orderTotal")
```

The **sumBy** and **sumByDouble** functions are effectively specific variations of **fold**. When you just need to add values, **sumBy** and **sumByDouble** omit the initial value and accumulator from your code – a nice win for readability. Also note that the **sumBy** functions do not need to be called on collections with numeric values. Here, you called it on a **List<String>** with no issues. As long as your lambda expression returns a numeric value, you can aggregate values using the **sumBy** or **sumByDouble** functions.

Much like how **associate** can simplify a **map** call followed by a call to **toMap**, **sumBy** reduces the complexity of calling **fold**. There are many functions that act as a shorthand or convenience function compared to the more generic **map**, **flatMap**, **filter**, and **fold** calls. Keep an eye out for them and check out the API reference for a full list of functional programming operations you can use. **List**'s APIs, for example, can be found at kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/.

For the More Curious: The vararg Keyword

As you have seen, the collection-builder functions like **listOf** and **mapOf** accept the desired contents of a collection as a parameter. You can pass in any number of arguments to these functions. But all the other functions you have declared have had a fixed number of parameters; you cannot pass in an extra argument, because Kotlin will not know what to do with it. So how do the collection builders work?

The collection-builder functions – as well as several other functions in the standard library – accept a variable number of arguments. They specify this using the `vararg` keyword, which you can see in the full signature of the **listOf** function, shown below. (There is also some unfamiliar syntax involving `<T>`, which you will learn about in [Chapter 18](#).)

```
public fun <T> listOf(vararg elements: T): List<T> = ...
```

`vararg` parameters can have zero, one, or many arguments specified wherever they are called. Inside the body of the **listOf** function, the type of `elements` is actually an **Array**, which you might recall from [the section called For the More Curious: Array Types](#). The number of items in `elements` can be queried, and you can also iterate over the items or access an item at a particular index, much like you can with the collection types.

It is also possible to declare your own `vararg` parameters. Try it out for yourself in the REPL:

Listing 11.14 Printing a variable number of messages (REPL)

```
fun printAll(vararg messages: String) {  
    println("I have ${messages.size} things to say.")  
    messages.forEach { println(it) }  
}  
  
printAll("Hello, World!", "Madrigal has left the building.")  
I have 2 things to say.  
Hello, World!  
Madrigal has left the building.
```

In practice, you will see `vararg` parameters in the standard library more frequently than you will use them in your own code. But it is helpful to be aware of this behavior to understand how the collection builders work. It also offers a convenient syntax if you find yourself declaring a function that accepts a group of items and you do not want to manually wrap them in a collection type.

For the More Curious: Arrow.kt

In this chapter you saw some of the functional programming-style tools that are included in Kotlin’s standard library, like **map**, **flatMap**, and **filter**.

Kotlin is a “multiparadigm” language, meaning it mixes the styles of object-oriented, imperative, and functional programming. If you have worked with a strictly functional programming language like Haskell, you know that it offers useful functional programming ideas that go further than the basics included in Kotlin.

For example, Haskell includes the **Maybe** type – a type that includes support for either something or an error – and allows operations that might result in an error to be represented using a type instead. Using a **Maybe** type allows you to represent an exception, like incorrectly parsing a number, without throwing an exception – which allows you to not need try/catch logic in your code.

Representing an exception without dealing with try/catch logic is a good thing. Some view try/catch as a form of GOTO statement: More often than not, it leads to code that is difficult to read and maintain.

Many of the functional programming features found in Haskell can be brought to Kotlin through libraries like Arrow.kt (<http://arrow-kt.io/>).

For example, the Arrow.kt library includes a flavor of the **Maybe** type found in Haskell called **Either**. Using **Either**, it is possible to represent an operation that could result in failure without resorting to throwing exceptions and try/catch logic.

Consider, for example, a function that parses some user input from a string to an **Int**. If the value is a number, it should be parsed as an **Int**, but if it is invalid, it should instead be represented as an error.

Using **Either**, the logic would read as follows:

```
fun parse(s: String): Either<NumberFormatException, Int> =
    if (s.matches(Regex("-?[0-9]+"))) {
        Either.Right(s.toInt())
    } else {
        Either.Left(NumberFormatException("$s is not a valid integer."))
    }

val x = parse("123")

val value = when(x) {
    is Either.Left -> when (x.a) {
        is NumberFormatException -> "Not a number!"
        else -> "Unknown error"
    }
    is Either.Right -> "Number that was parsed: ${x.b}"
}
```

No exceptions, no try/catch blocks – just easy-to-follow logic.

Challenge: Reversing the Values in a Map

Using the functional techniques you learned in this chapter, write a function called **flipValues** that allows you to flip-flop the keys and values in a map. For example:

```
val gradesByStudent = mapOf("Josh" to 4.0, "Alex" to 2.0, "Jane" to 3.0)
{Josh=4.0, Alex=2.0, Jane=3.0}

flipValues(gradesByStudent)
{4.0=Josh, 2.0=Alex, 3.0=Jane}
```

Challenge: Finding the Most-Liked Menu Item

Right now, Taernyl picks the item of the day randomly from a list of favorite items. As long as one of his patrons likes an item, it is eligible to be chosen as the item of the day. Taernyl would like the item of the day to be the true favorite item of all his patrons.

Update the code that determines the item of the day to pick the menu item that has the most occurrences in the patrons' favorites. If there is a tie, you can return either menu item. For a more difficult challenge, try to perform this computation without declaring any new variables. (Hint: You may find the **fold** and **maxOf** functions useful. Also, consider what kind of collection would be useful for counting the occurrences of each menu item.)

Scope Functions

Scope functions are general utility functions in the Kotlin standard library that can help you write more expressive and concise code. In this chapter you will meet the six most commonly used scope functions – **apply**, **let**, **run**, **with**, **also**, and **takeIf** – and see examples of what they can do.

We will begin by introducing each of these extension functions, and then you will incorporate one of them into NyetHack.

Every scope function is called on a value, commonly known as the *receiver*, and takes in a lambda that defines the work you want to do with the value. The term receiver stems from the fact that Kotlin's scope functions are *extension functions* under the hood, and *receiver* is the term for the subject of an extension function. You will learn about extension functions, which are a flexible way to define functions on types, in [Chapter 19](#).

apply

First on our tour of common scope functions is **apply**. **apply** can be thought of as a configuration function: It allows you to call a series of functions on a receiver to configure it for use. After the lambda provided to **apply** executes, **apply** returns the configured receiver.

apply can be used to reduce the amount of repetition when configuring an object for use. For example, you might want to create complicated rules to determine what elements appear in your collections. In NyetHack, for instance, Taernyl’s patrons might not behave randomly: The citizens of Kronstadt might have routines that determine when they visit the tavern.

One way to implement these complex rules would be to create a **MutableList** and populate the list manually. If you wanted to end up with a read-only **List** instead of a **MutableList**, you would also need to declare a second variable, like this:

```
val patrons: MutableList<String> = mutableListOf()
if (isAfterMidnight) { patrons.add("Sidney") }
if (isOpenMicNight) { patrons.add("Janet") }
if (isHappyHour) { patrons.add("Jamie") }
if (patrons.contains("Janet") || patrons.contains("Jamie")) { patrons.add("Hal" ) }

val guestList: List<String> = patrons.toList()
```

Using **apply**, the same configuration can be achieved with less repetition and without the `patrons` variable:

```
val guestList: List<String> = mutableListOf<String>().apply {
    if (isAfterMidnight) { add("Sidney") }
    if (isOpenMicNight) { add("Janet") }
    if (isHappyHour) { add("Jamie") }
    if (contains("Janet") || contains("Jamie")) { add("Hal" ) }
}.toList()
```

apply allows you to drop the variable name from every function call performed to configure the receiver. This is because **apply** *scopes* each function call within the lambda to the receiver it is called on.

This behavior is sometimes referred to as [relative scoping](#), because all the function calls within the lambda are now called relative to the receiver. Another way to say this is that they are [implicitly called](#) on the receiver.

let

Another commonly used scope function is **let**, which you encountered in [Chapter 7](#). **let** scopes a variable to the lambda provided and passes the receiver as an argument. This lets you refer to the receiver with the `it` identifier, which you learned about in [Chapter 8](#).

Since Madrigal is new to the town, she is likely to catch some attention. A patron in the tavern – possibly the first patron that sees her – may come up to greet her and give Madrigal an introduction. You can form this introduction using the **let** function:

```
val patrons: List<String> = listOf(...)
val greeting = patrons.first().let {
    "$it walks over to Madrigal and says, \"Hi! I'm $it. Welcome to Kronstadt!\""
}
```

Without **let**, you would need to assign the first element to a variable to remember which patron is talking:

```
val patrons: List<String> = listOf(...)
val friendlyPatron = patrons.first()
val greeting = "$friendlyPatron walks over to Madrigal and says, \"Hi! \" +
    "I'm $friendlyPatron. Welcome to Kronstadt!\""
```

When combined with other Kotlin syntax, **let** provides additional benefits. You saw **let** for the first time in [Chapter 7](#) as part of a mechanism for handling null safety:

```
censoredQuest?.let {
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |"$censoredQuest"
        |""".trimMargin()
    )
}
```

Consider the example above where a friendly patron will greet Madrigal. Depending on the time of day, Madrigal might walk into an empty tavern. To account for this, you can use **firstOrNull** instead of **first** to handle this edge case without crashing.

```
val patrons: List<String> = listOf(...)
val greeting = patrons.firstOrNull()?.let {
    "$it walks over to Madrigal and says, \"Hi! I'm $it. Welcome to Kronstadt!\""
} ?: "Nobody greets Madrigal because the tavern is empty"
```

Using the safe call operator (`?.`) means that **let** executes if and only if the receiver is not null. In this case, that means **let** only executes when **firstOrNull** returns a patron name. By using **let** with a safe call, you ensure that the `it` argument will be non-null, and you can safely perform operations inside the lambda expression without checking its nullity again.

Compare the example above using **let** with the version below that does not use **let**:

```
val patrons: List<String> = listOf(...)
val friendlyPatron = patrons.firstOrNull()
val greeting = if (friendlyPatron != null) {
    "$friendlyPatron walks over to Madrigal and says, \"Hi! \" +
    "I'm $friendlyPatron. Welcome to Kronstadt!\""
} else {
    "Nobody greets Madrigal because the tavern is empty"
}
```

This version is functionally equivalent, but slightly more verbose. The `if/else` structure uses the full `friendlyPatron` variable three times: once in the condition and twice to create the resulting string. **let**, on the other hand, allows a fluent or [chainable](#) style that lets you omit the declaration of intermediate variables.

Keep in mind that **let** on its own is not a null safety technique. It is the combination of a scoping function and the safe call that provides null safety.

You can also use **apply** with a safe call, but it is less common. When reaching for a null safety pairing, most Kotlin programmers choose **let** – although others prefer **run**, the next scope function on our list, instead. They are similar, but you may find yourself preferring one over the other.

let can be called on any kind of receiver and returns the result of evaluating the lambda you provide. In the example above, **let** is called on a string, `patrons.firstOrNull()`. The lambda passed to **let** accepts the receiver it is called on as its only argument. You can therefore access the argument using the `it` identifier.

Several differences between **let** and **apply** are worth mentioning: As you saw, **let** passes the receiver to the lambda you provide, but **apply** passes nothing. Also, **apply** returns the current receiver when the lambda completes. **let**, on the other hand, returns the last line of the lambda (the [lambda result](#)).

run

Next up on our tour of the scope functions is **run**. **run** provides the same relative scoping behavior as **apply**, and it returns the result of the lambda expression, instead of the receiver itself, like **let**.

Say you wanted to keep track of music that is playing in the tavern. You might write code that looks like this:

```
val tavernPlaylist = mutableListOf("Korobeiniki", "Kalinka", "Katyusha")
val nowPlayingMessage: String = tavernPlaylist.run {
    shuffle()
    "${first()} is currently playing in the tavern"
}
```

The **shuffle** function is implicitly performed on the receiver – the **List** instance. This is just like the **add** and **contains** functions you saw with **apply**. However, unlike **apply**, **run** returns the lambda result – here, a message saying which Russian folk song is currently playing in the tavern.

By the way, there is a second flavor of **run** that is not called on a receiver. This form is far less commonly seen, but we include it here for completeness:

```
val healthPoints = 90
val healthStatus = run {
    if (healthPoints == 100) "perfect health" else "has injuries"
}
```


with

with is very similar to **run**. It behaves identically, but it uses a different calling convention. Unlike the scope functions you have seen so far, **with** requires its argument to be accepted as the first parameter rather than calling the scope function on a receiver type:

```
val nameTooLong = with("Polarcubis, Supreme Master of NyetHack") {  
    length >= 20  
}
```

Up to this point, your scope functions have been called on the value itself – as in, `"Polarcubis".run { ... }`. But **with** breaks this convention. You call with *first*, like `with("Polarcubis") { ... }`.

Because of this difference, we recommend using **run** instead of **with** – it is more consistent, more prevalent, and oftentimes more readable. But the decision between **run** and **with** is a stylistic one. Depending on the context, you may prefer to use **run** or **with**:

```
val player: Player = ...  
val monster: Goblin = ...  
  
// Both calls compile as `player.fight(monster)`  
with(player) { fight(monster) }  
player.run { fight(monster) }
```

These calls work the same way and return the same result.

also

The **also** function works very similarly to the **let** function. Just like **let**, **also** passes the receiver you call it on as an argument to the lambda you provide. But there is one major difference between **let** and **also**: **also** returns the receiver, rather than the result of the lambda – just like **apply** does.

This makes **also** especially useful for adding multiple side effects from a common source. In the example below, **also** is called twice to organize two different operations: One prints the filename, and the other assigns a variable, `fileContents`, with the contents of the file.

```
var fileContents: List<String>
File("file.txt")
    .also { print(it.name) }
    .readLines()
    .also { fileContents = it }
```

Because **also** returns the receiver instead of the result of the lambda, you can continue to chain additional function calls onto the original receiver.

takeIf

The last stop on our tour of the scope functions is **takeIf**. **takeIf** works differently than the other scope functions: It evaluates a predicate condition provided in a lambda that returns either true or false. If the condition evaluates as true, the receiver is returned from **takeIf**. If the condition is false, null is returned instead.

Consider the following example, which reads a file if and only if it exists.

```
val fileContents = File("myfile.txt")
    .takeIf { it.exists() }
    ?.readText()
```

Without **takeIf**, this would be more verbose:

```
val file = File("myfile.txt")
val fileContents = if (file.exists()) {
    file.readText()
} else {
    null
}
```

The **takeIf** version does not require the temporary variable **file**, nor does it need to specify the null return value. **takeIf** is useful for checking that some condition required for assigning a variable or proceeding with work is true before continuing. Conceptually, **takeIf** is similar to an `if` statement, but with the advantage of being directly callable on an instance, often allowing you to remove a temporary variable assignment.

We said that the tour was over, but there is a complementary function to **takeIf** that we should mention for completeness: **takeUnless**. The **takeUnless** function is exactly like **takeIf** except that it returns the original value if the condition you define is *false*. This example reads the file if it is not hidden (and returns null otherwise):

```
val fileContents = File("myfile.txt").takeUnless { it.isHidden }?.readText()
```

We recommend that you prefer **takeIf** – especially for complex conditions – since it often makes your conditions more readable. Compare the “understandability” of these two phrases:

- “Return the value if the condition is true” – **takeIf**
- “Return the value unless the condition is true” – **takeUnless**

If you found yourself having to pause slightly for the second phrase, you are like us: **takeUnless** seems to be a less natural way of describing the logic you want to express.

For simple conditions (as in the example above), **takeUnless** is not problematic. But with more complicated examples, we find **takeUnless** harder to parse. We recommend limiting your usages of **takeUnless** to very short conditions that you would otherwise negate inside your predicate’s lambda expression.

Using Scope Functions

[Table 12.1](#) summarizes the Kotlin scope functions discussed in this chapter:

Table 12.1 Scope functions

Function	Passes receiver to lambda as argument?	Provides relative scoping?	Returns
let	Yes	No	Lambda result
apply	No	Yes	Receiver
run ^a	No	Yes	Lambda result
with ^b	No	Yes	Lambda result
also	Yes	No	Receiver
takeIf	Yes	No	Nullable version of receiver
takeUnless	Yes	No	Nullable version of receiver

^a The non-receiver version of **run** (less commonly used) passes no receiver, performs no relative scoping, and returns the lambda result.

^b **with** is not called on the receiver like this: "hello.with { .. }". Instead, it treats the first argument as the receiver, the second being the lambda, like this: with("hello") { .. }. It is the only scope function that works this way, and it is used less often than the other scope functions.

As their name suggests, scope functions are best used when you want to temporarily create a new scope or change the scope that your program is running in. Any time you would use a temporary variable might be a good time to consider using a scope function.

Think about NyetHack. Are there any places you can think of that are begging for the scope function treatment? Because of your utilization of Kotlin’s functional programming operators, there is not much that could be cleaned up with a scope function.

But take a look at your departing patron logic in `Tavern.kt`:

```
val departingPatrons: List<String> = patrons
    .filter { patron -> patronGold.getDefault(patron, 0.0) < 4.0 }
patrons -= departingPatrons
patronGold -= departingPatrons
departingPatrons.forEach { patron ->
    narrate("$heroName sees $patron departing the tavern")
}
```

This block of code uses `departingPatrons` multiple times, and you can make it look nicer using scope functions. If you wanted to, you could find a way to use any of the scope functions introduced in this chapter. But we recommend using **also** to encapsulate your side effects. Open `Tavern.kt` and try it out:

Listing 12.1 Using also (Tavern.kt)

```
... visitTavern() {
    ...
    displayPatronBalances(patronGold)

    val departingPatrons: List<String> = patrons
    patrons
    .filter { patron -> patronGold.getDefault(patron, 0.0) < 4.0 }
patrons -= departingPatrons
patronGold -= departingPatrons
    .also { departingPatrons ->
        patrons -= departingPatrons
        patronGold -= departingPatrons
    }
departingPatrons.forEach { patron ->
    .forEach { patron ->
        narrate("$heroName sees $patron departing the tavern")
    }

    narrate("There are still some patrons in the tavern")
    narrate(patrons.joinToString())
}
...
```

Run NyetHack and confirm that your program behaves the same.

This change lets you consolidate all the departing patron logic into a single statement. You also were able to completely remove the `departingPatrons` variable, since you did not need to access the variable later on.

In this chapter, you saw how to simplify your code using scope functions. They give you the ability to write code that is not only concise but also has the unique feel of Kotlin. We will use scope functions in the rest of this book

where applicable.

In this part of the book, you saw how to use Kotlin's collection types and gained experience with the functional programming techniques that Kotlin provides. In the next set of chapters, you will switch gears and start learning a different programming paradigm: object-oriented programming.

Part IV

Object-Oriented Programming

In the next five chapters, you will learn about another programming paradigm called object-oriented programming. The object-oriented programming paradigm has been around since the 1960s and continues to be popular because it provides a set of useful tools for simplifying the structure of a program. It is a great technique for organizing portions of your code into reusable and extensible components, called *classes* and *objects*.

You will learn how to define classes, initialize objects, make your classes inherit from other classes and interfaces, and leverage special types of classes in Kotlin, like singletons and data classes. At the end of this part of the book, NyetHack will be a fully interactive game, with rooms to explore and monsters to battle.

13

Classes

Central to the object-oriented programming style are [classes](#), definitions of the unique categories of “things” your code represents. Classes define what sort of data those things will consist of and what kind of work they can do.

To make NyetHack object-oriented, you will start by identifying the unique types of things that will exist in the program’s world and defining classes for them. In this chapter, you will add a custom **Player** class to NyetHack, which you will use to represent a NyetHack player’s particular characteristics.

Defining a Class

A class can be defined in its own file or alongside other elements, like functions or variables. Defining a class in its own file gives it room to grow as the program scales up over time, and that is what you will do in NyetHack. Create a new `Player.kt` file: right-click the `src/main/kotlin` folder, choose `New → Kotlin Class/File`, and choose `File` when you enter the name. Then declare your first class with the `class` keyword:

Listing 13.1 Defining the `Player` class (`Player.kt`)

```
class Player
```

A class is often declared in a file matching its name, but it does not have to be. You can define multiple classes in the same file – and you may want to if you have multiple classes used for a similar purpose.

With that, your class is defined. All that is left is to give it some work to do.

Constructing Instances

A class declaration is like a blueprint. Blueprints contain the details for how to construct a building, but they are not a building. Your **Player** class declaration works similarly: So far, a player has not been constructed – you have only created the (so far, quite sparse) blueprint.

When you start a new game of NyetHack, the **main** function is called, and one of the first things that you will want to do is create a player character to play the game. To construct a player so it can be used in NyetHack, you must [instantiate](#) it – create an [instance](#) of it – by calling its [constructor](#).

In `NyetHack.kt`, where variables are declared in the **main** function, instantiate a **Player**:

Listing 13.2 Instantiating a Player (NyetHack.kt)

```
var heroName = ""
val player = Player()

fun main() {
    heroName = promptHeroName()

    // changeNarratorMood()
    narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
    visitTavern()
}
...
```

You called **Player**’s constructor by suffixing the **Player** class name with parentheses. This constructs an instance of the **Player** class. The `player` variable is now said to “contain an instance of the **Player** class.” Every class definition in Kotlin creates a corresponding type, so your `player` variable is of type **Player**.

A constructor does what its name says: It constructs. Specifically, it constructs an instance and prepares it for use. The syntax for calling a constructor is a lot like calling a function: It uses parentheses to capture arguments for its parameters. You will see other ways instances can be constructed in [Chapter 14](#).

Grouping the logic about the “things” in your code using classes keeps your code organized at scale. As NyetHack grows, you will add more classes, each with its own responsibilities.

Now that you have an instance of **Player**, what can you do with it?

Class Functions

Class definitions can specify two types of content: *behavior* and *data*. In NyetHack, a player should be able to take various actions: perform combat, move, cast a spell, or check their inventory, for example. You define behavior for a class by adding function definitions to its class body. Functions defined within a class are called [class functions](#).

You already have some player data defined in NyetHack. You will be moving that into the **Player** class and introducing a few behaviors and new pieces of data.

Define the first **Player** behavior by adding a function to allow the player to cast a spell:

Listing 13.3 Defining a class function (Player.kt)

```
class Player {
    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```

Here, you defined a [class body](#) for **Player** with a pair of curly braces. The class body holds definitions for the class's behavior and data, much like the actions of a function are defined within the function body.

Why define **castFireball** on **Player**? In NyetHack, summoning a glass of Fireball is something that a player does: It cannot happen without an instance of **Player**, and it is performed by the particular player on which **castFireball** is called. Defining **castFireball** as a class function, so it is called on an instance of the class, reflects this logic.

In NyetHack.kt, add a call to **castFireball** as a class function in **main**:

Listing 13.4 Calling a class function (NyetHack.kt)

```
var heroName = ""
val player = Player()

fun main() {
    heroName = promptHeroName()

    // changeNarratorMood()
    narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
    visitTavern()
    player.castFireball()
}
...
```

Run NyetHack and confirm that the player summons a round of Fireball in the tavern.

Visibility and Encapsulation

Adding behavior to a class with class functions (and data with class properties, as you will see in a moment) builds a description of what that class can do and be, and that description is visible to anyone with an instance of that class.

By default, any function or property without a visibility modifier is public – meaning it is accessible from any file or function in your program. Since you did not include a visibility modifier on **castFireball**, it can be called from everywhere.

In some cases, like with **castFireball**, you want other parts of your code to be able to access your class properties or call your class functions. But you might have other class functions or properties you do not want to be called from elsewhere in your codebase.

As the number of classes in your program grows, so does your codebase’s complexity. Hiding the implementation details that do not need to be visible from other parts of your codebase helps to ensure that the logic of your code is clear and concise. That is where visibility comes into play.

While a public class function can be invoked anywhere in the program, a private class function cannot be invoked outside the class on which it is defined. This idea of restricting visibility to certain class functions or properties drives a concept in object-oriented programming known as [encapsulation](#). Encapsulation says that a class should selectively expose functions and properties to define how other objects interact with it. Anything that is not essential to expose, including implementation details of exposed functions and properties, should be kept private.

For example, although **castFireball** is called from **main**, **main** does not care about how **castFireball** is implemented. It only cares that a glass of Fireball is summoned. So while the function itself may be exposed, the details of its implementation should not matter to the caller.

In short: When building classes, expose only what you need to.

[Table 13.1](#) lists the available visibility modifiers in Kotlin:

Table 13.1 Visibility modifiers

Modifier	Description
public (default)	The function or property will be accessible by code outside the class. By default, functions and properties without a visibility modifier are public.
private	The function or property will be accessible only within the same class.
protected	The function or property will be accessible only within the same class or its subclass.
internal	The function or property will be accessible within the same module.

We will discuss protected visibility in [Chapter 15](#).

These visibility modifiers are the same no matter which platform your Kotlin code targets. If you are familiar with Java, you may have noticed that the package private visibility level is not available in Kotlin. We will explain why in [the section called *For the More Curious: Package Private*](#) at the end of this chapter.

Class Properties

Class function definitions describe the behavior associated with a class. Data definitions, better known as [class properties](#), are the attributes required to represent the specific state or characteristics of a class. For example, **Player**’s class properties could represent a player’s name, current health points, fantasy race, alignment, gender, and other attributes.

Currently, you define a name for a player in the **main** function, but your new class definition is a better place for it. Update `Player.kt` with a **name** property. (The value for **name** may look sloppy, but there is a method to our madness – enter it as shown.)

Listing 13.5 Defining the name property (Player.kt)

```
class Player {
    val name = "madrigal"

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```

You add the **name** property to the **Player** class body, including it as relevant data a **Player** instance contains. **name** is defined as a `val`; like variables, properties can represent either read-only or mutable data using the `val` and `var` keywords, respectively. We will talk more about property mutability later in this chapter.

When an instance of a class is constructed, all its properties must have values. This means that, unlike other variables, class properties must be assigned an initial value. For example, this code is invalid, because **name** is not assigned at declaration:

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

We will explore the nuances of class and property initialization in [Chapter 14](#).

Next, remove the `heroName` declaration from `NyetHack.kt`:

Listing 13.6 Removing heroName from main (NyetHack.kt)

```
var heroName: String = ""
val player = Player()

fun main() {
    heroName = promptHeroName()

    // changeNarratorMood()
    narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
    narrate("${player.name}, ${createTitle(player.name)}, heads to the town square")

    visitTavern()
    player.castFireball()
}
...
```

Now that **name** is a property of **Player**, you use dot syntax to access the `player` variable’s **name** property when calling **narrate**. This [dot syntax](#) is how you read and write properties and invoke functions for instances of an object.

If you were to build your program right now, you would get several compiler errors in `Tavern.kt`, each saying “Unresolved reference: `heroName`.” Open `Tavern.kt` and fix the errors by replacing references to `heroName` with `player.name`.

Listing 13.7 Resolving references to Player’s name property (Tavern.kt)

```
...
fun visitTavern() {
    narrate("heroName{player.name} enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    narrate(menuItems.joinToString())
    ...
    val patronGold = mutableMapOf(
        TAVERN_MASTER to 86.00,
        heroNameplayer.name to 4.50,
        *patrons.map { it to 6.00 }.toTypedArray()
    )

    narrate("heroName{player.name} sees several patrons in the tavern:")
    narrate(patrons.joinToString())
    ...
    patrons.filter { patron -> patronGold.getOrDefault(patron, 0.0) < 4.0 }
        .also { departingPatrons ->
            patrons -= departingPatrons
            patronGold -= departingPatrons
        }
        .forEach { patron ->
            narrate("heroName{player.name} sees $patron departing the tavern")
        }
    ...
}
...
private fun displayPatronBalances(patronGold: Map<String, Double>) {
    narrate("heroName{player.name} intuitively knows how much money each patron has")
    patronGold.forEach { (patron, balance) ->
        narrate("$patron has ${"%0.2f".format(balance)} gold")
    }
}
}
```

Run `NyetHack`. Madrigal visits the tavern just as before, but now you access the **name** property from the instance

of the **Player** class rather than from a top-level variable in `NyetHack.kt`.

Later in this chapter, you will refactor `NyetHack` to move the other data belonging to the **Player** class into the class definition.

Property getters and setters

Properties model the characteristics of each instance of a class. They also provide a way for other entities to interface with the data the class keeps track of, represented in a compact and concise syntax. This interaction happens through getters and setters.

For each property you define, Kotlin will generate up to three components: a [field](#), a [getter](#), and a [setter](#). A field is where the data for a property is stored. You cannot directly define a field on a class. Kotlin encapsulates the fields for you, protecting the data in the field and exposing it via getters and setters.

A property's getter specifies how the property is read. Getters are generated for every property. A setter defines how a property's value is assigned, so it is generated only when a property is writable – in other words, when the property is a `var`.

Imagine that you are in a restaurant where the menu advertises spaghetti, among other foods. You order spaghetti, and the waiter serves you spaghetti dressed up with spaghetti sauce and cheese. You do not have access to the kitchen, and the waiter handles everything behind the scenes for you, even adding spaghetti sauce and cheese to your order of spaghetti. You are like the caller, and the waiter is the getter.

As a patron of this restaurant, you do not want the responsibility of boiling water when you order spaghetti. Rather, you simply want to order spaghetti and have it brought to you. And the restaurant does not want you in the kitchen, nosing around in the ingredients and putting together dishes in your own way. This is encapsulation at work.

Although default getters and setters are provided automatically by Kotlin, you can change the behavior of the getters and setters when you want to specify how the data will be read or written. To do this, you write a custom getter or setter.

To see how custom getters work, define a getter for `name` that ensures that the first letter in this string is capitalized when the name is accessed.

Listing 13.8 Defining a custom getter (`Player.kt`)

```
class Player {
    val name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```

When you define a custom getter for a property, you change how the property works when it is accessed. Because `name` contains a proper noun, you always want it to be capitalized when you reference it. This custom getter makes sure of that.

Run `NyetHack` and confirm that `Madrigal` now prints with a capital M.

The `field` identifier here points to the backing field that Kotlin manages for your property automatically. The backing field is the data that the getters and setters use to read and write the data that represents the property. It is like the ingredients in the restaurant kitchen – the caller never sees the backing field directly, only the data as presented by the getter. In fact, you can only access a property's field inside a custom getter or a setter.

When the capitalized version of `name` is returned, the backing field is not modified. If the value assigned to `name` is not capitalized, as in your code, it remains lowercase after the getter does its work.

A setter, on the other hand, *does* modify the backing field of the property on which it is declared. Add a setter to `name` that uses the `trim` function to remove any leading and trailing spaces from the value it is passed.

Listing 13.9 Defining a custom setter (`Player.kt`)

```
class Player {
    val name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    set(value) {
        field = value.trim()
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```

There is a problem with adding a setter to this property, which IntelliJ is warning you about ([Figure 13.1](#)):

Figure 13.1 `val` properties are read-only

```
class Player {
    val name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    set(value) {
        field = value.trim()
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```



Because you defined the `name` property as a `val`, it is read-only and cannot be modified, even with a setter. This protects your `vals` from being modified without your consent.

IntelliJ's complaint underscores an important point about setters: They are triggered when the value of a property is set. It is not logical (and, in fact, it is an error) to define a setter for a `val` property; since the value is read-only, the setter could never do its job.

You want to be able to change the player's name, so change the `name` property from a `val` to a `var`. (From this point forward, we will show all changes to code inline when possible.)

Listing 13.10 Making name mutable (`Player.kt`)

```
class Player {
    val var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    set(value) {
        field = value.trim()
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```

Now, `name` can be modified according to the rules outlined in its custom setter, and IntelliJ's warnings disappear accordingly.

Property getters are called using the same access syntax as the other variables that you have seen. Property setters are called using the assignment operator that you have used to assign values to variables.

In the Kotlin REPL, try changing a player's name from outside the **Player** class. You will first need to reload the

REPL with the  Build and restart button to its left so the change to **Player** is recognized. (Make sure to include a space in the name string, as shown, to give the setter something to do.)

Listing 13.11 Changing a player's name (REPL)

```
val player = Player()
player.name = "estragon "
print(player.name + "TheBrave")
EstragonTheBrave
```

Here you can see the effect of both the getter and the setter on the new value for `name`.

Assigning new values to class properties changes the state of the class on which they are assigned. If `name` were still a `val`, then the example that you just tried in the REPL would result in the error message `error: val cannot be reassigned`.

Property visibility

Properties are different from variables defined locally within a function. When a property is defined, it is defined at the class level. As such, it may be accessible to other classes, if its visibility allows. Over-permissive visibility can cause problems: If other classes have access to a **Player**'s data, then any class in your application could make changes to that instance of **Player** at will.

Properties provide fine-grained control around reading and modifying data through their getters and setters. All properties have getters – and all `var` properties have setters – whether you define custom behavior for them or not. By default, the visibility of a property's getter and setter match the visibility of the property itself. So if you have a public property, both its getter and setter are public.

What if you want to expose access to a property but do not want to expose its setter? You can define the visibility of the setter separately by making the property’s setter private:

Listing 13.12 Hiding name’s setter (Player.kt)

```
class Player {
    var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    private set(value) {
        field = value.trim()
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```

Now, `name` can be accessed from anywhere in `NyetHack`, but it can only be modified from within **Player**. This technique is quite useful if you want to control whether certain properties can be modified by other parts of your application.

A setter’s visibility cannot be more permissive than the property it is defined on. Visibility modifiers are designed to restrict who can access which behaviors of your properties, and Kotlin does not allow a more permissive visibility modifier to be used on a getter or setter than the property’s own visibility modifier.

You can also apply a visibility modifier to a setter without defining your own set behavior. To do this, omit the parentheses and curly braces after the set keyword, like this:

```
class Player {
    var name = "madrigal"
    private set
    ...
}
```

As before, this would ensure that while `name` could be accessed from anywhere in the program, it could only be changed from within the **Player** class – but without specifying any additional behavior for changes. This concise syntax is useful when you want to prevent external modifications to a variable but do not need to apply your own logic to the setter.

As for getter visibility, you cannot use visibility modifiers to make a getter’s visibility different from the property’s. (You are allowed to specify a visibility modifier for a getter, but it must match the property’s visibility – so, in practice, it is not useful to do so.)

Computed properties

Earlier, we said that when you define a property, a field is generated to store the value the property encapsulates. That is true ... except in a particular case: [computed properties](#). A computed property is a property that is specified with a custom getter – and setter if the property is a var – in a way that makes a field unnecessary. In such cases, Kotlin will not generate a field.

Consider your **createTitle** function in `NyetHack.kt`. This function creates a designation for the player and is dependent on their name; it should be in the **Player** class. But while you could just move the function into **Player**, consider this: The title is data – not behavior. So it really belongs in a property – but it needs to be able to respond to changes in the player’s name. Using a computed property will allow you to keep this value in a property and ensure it is always up to date.

Add a computed property for the player’s `title` as well as a new function to allow the player to change their name.

Listing 13.13 Adding title to Player (Player.kt)

```
class Player {

    var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    private set(value) {
        field = value.trim()
    }

    val title: String
    get() = when {
        name.all { it.isDigit() } -> "The Identifiable"
        name.none { it.isLetter() } -> "The Witness Protection Member"
        name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowels"
        else -> "The Renowned Hero"
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }

    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        name = newName
    }
}
```

Take your new computed property for a spin in `NyetHack.kt`, both before and after Madrigal submits the required paperwork to change her name.

Listing 13.14 Using computed properties (NyetHack.kt)

```
val player = Player()

fun main() {
    narrate("${player.name} is ${player.title}")
    player.changeName("Aurelia")
    // changeNarratorMood()
narrate("${player.name}, ${createTitle(player.name)}, heads to the town square")
    narrate("${player.name}, ${player.title}, heads to the town square")

    visitTavern()
    player.castFireball()
}

private fun promptHeroName(): String {
    ...
}

private fun createTitle(name: String): String {
    return when {
        name.all { it.isDigit() } -> "The Identifiable"
        name.none { it.isLetter() } -> "The Witness Protection Member"
        name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowels"
        else -> "The Renowned Hero"
    }
}
}
```

Run NyetHack. Your output will now look like this:

```
Madrigal is The Renowned Hero
Madrigal legally changes their name to Aurelia
Aurelia, The Master of Vowels, heads to the town square
Aurelia enters Taernyl's Folly
...
```

The value of `title` is computed each time the property is accessed. It has no initial or default value – and no backing field to hold a value. If the player’s name has changed, the value will automatically update so the title stays in sync with the player’s name.

You will look more carefully at how `val` and `var` properties are implemented and compiled in [the section called *For the More Curious: A Closer Look at var and val Properties*](#) near the end of this chapter.

In the next chapter, you will add more ways to instantiate **Player** as you learn about initialization. But before growing your application further, it is a good time to learn about packages.

Using Packages

A package is like a folder for similar elements that helps give a logical grouping to the files in your project. For example, the `kotlin.collections` package contains classes to create and manage lists and sets. Packages allow you to organize your project as it becomes more complex, and they also prevent naming collisions.

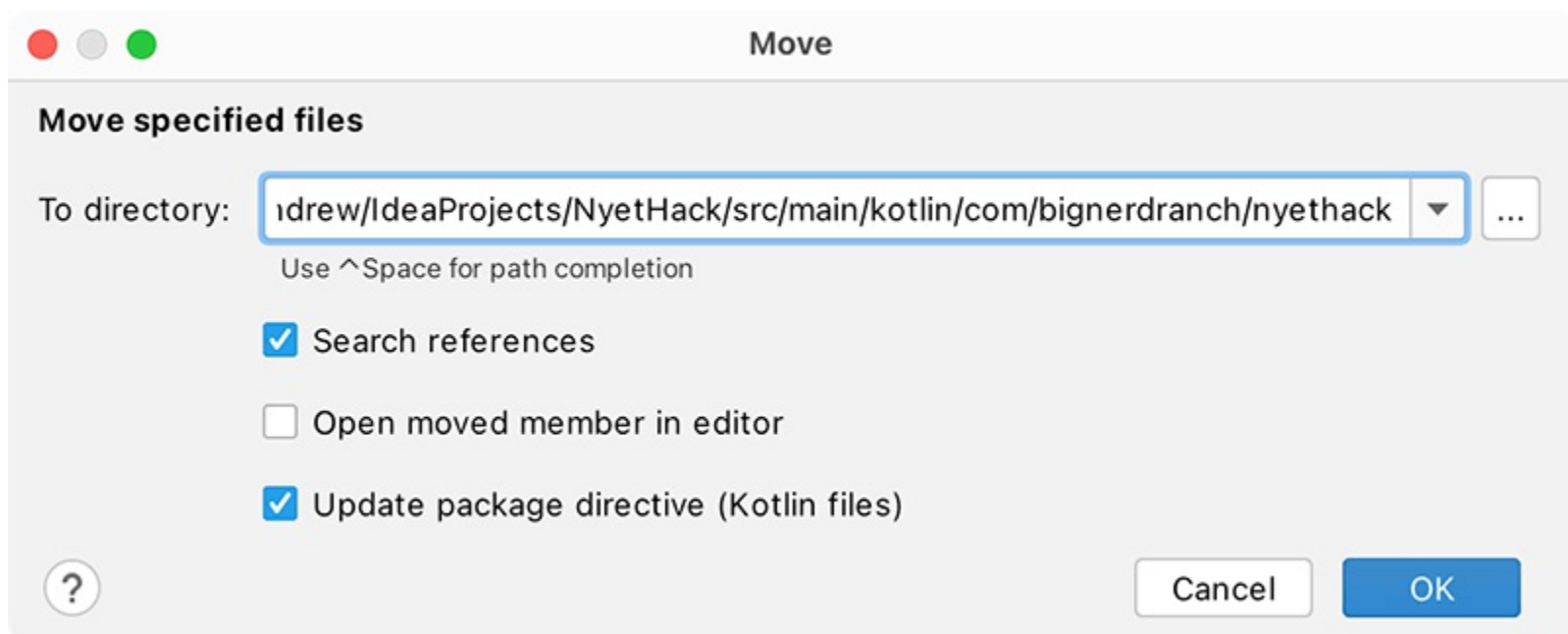
Create a package by right-clicking your `src/main/kotlin` directory and selecting `New → Package`. When prompted, name your package `com.bignerdranch.nyethack`. (You can name a package anything you like, but we prefer this reverse-DNS style that scales with the number of applications that you write.)

The package you created, `com.bignerdranch.nyethack`, is the top-level package for NyetHack. Including your files within a top-level package will prevent any naming collisions with classes that you define and classes defined elsewhere – for instance, in external libraries or modules. As you add more files, you can create additional packages to keep your code organized.

The new `com.bignerdranch.nyethack` package (which resembles a folder) is displayed in the project tool window. You can now move all your Kotlin code into this new package.

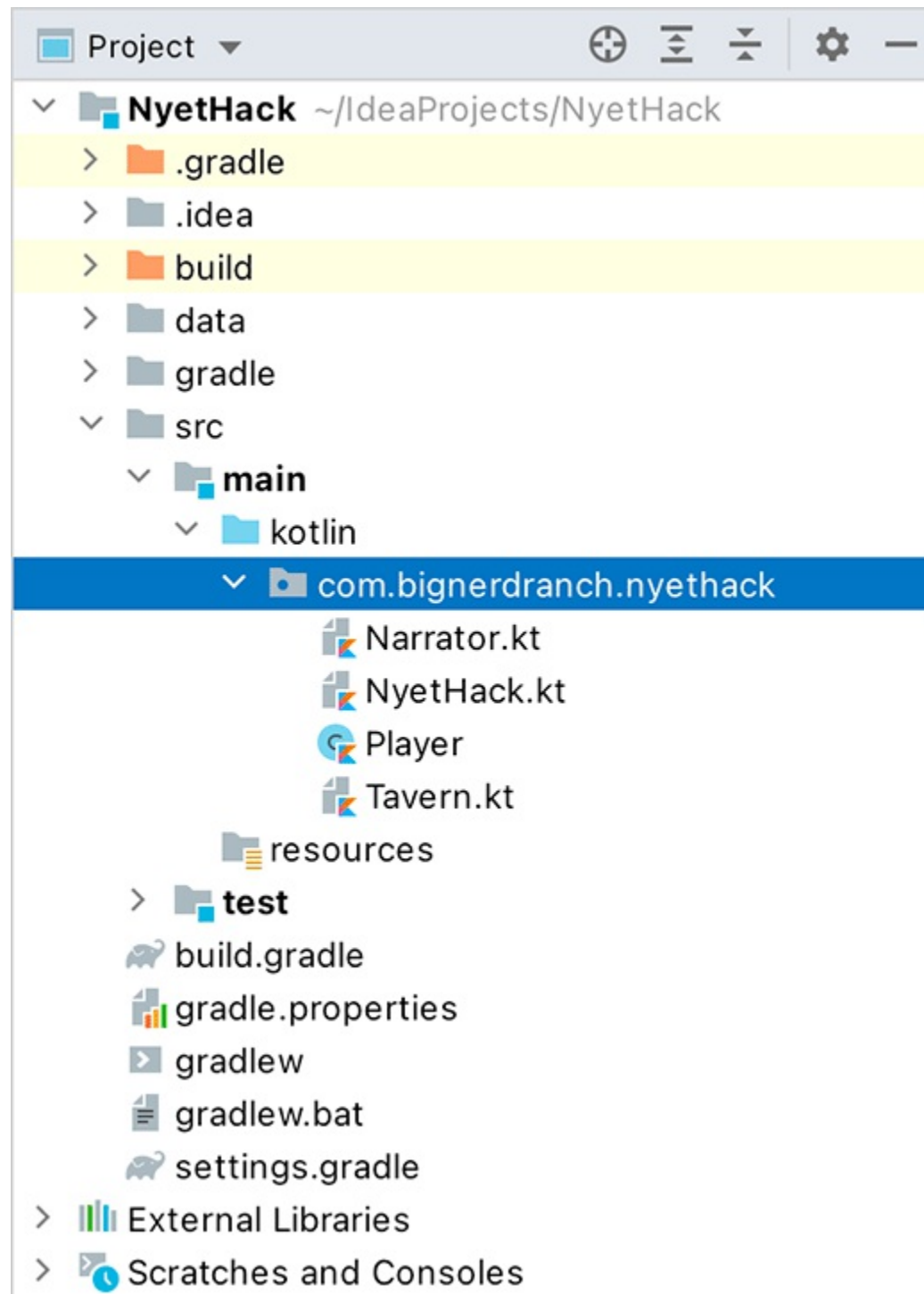
Select all your source files by Command-clicking (Ctrl-clicking) `Narrator.kt`, `NyetHack.kt`, `Player.kt`, and `Tavern.kt`. Drag the files into the `com.bignerdranch.nyethack` package. A Move dialog will appear ([Figure 13.2](#)). Ensure that Update package directive is enabled and click OK.

Figure 13.2 The Move refactoring dialog



After the operation completes, all your Kotlin files will be nested under the `com.bignerdranch.nyethack` package, as shown in [Figure 13.3](#). On disk, IntelliJ has created a `com/bignerdranch/nyethack` folder that now contains all your Kotlin code. Although there are three nested folders, IntelliJ’s project tool window simplifies this hierarchy to a single folder for easier navigation.

Figure 13.3 The `com.bignerdranch.nyethack` package



When you made this refactor, IntelliJ also inserted a line at the top of each file that reads `package com.bignerdranch.nyethack`. This line tells Kotlin which package the file belongs to. And although this package declaration does not have to match the folder structure where the file is saved on disk, we strongly encourage you to follow this convention.

Organizing code using classes, files, and packages will help ensure that your code is clear as your application grows in complexity.

For the More Curious: A Closer Look at var and val Properties

In this chapter you learned that the `var` and `val` keywords are used when specifying a class property – `var` for writable, and `val` for read-only.

You may be wondering how a Kotlin class property works under the hood. To understand how class properties are implemented, it is helpful to look at the decompiled bytecode – specifically, to compare the bytecode generated for a property depending on how it is specified. Take a look at `Player.kt`, which we have copied below:

```
package com.bignerdranch.nyethack

class Player {

    var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    private set(value) {
        field = value.trim()
    }

    val title: String
    get() = when {
        name.all { it.isDigit() } -> "The Identifiable"
        name.none { it.isLetter() } -> "The Witness Protection Member"
        name.count { it.lowercase() in "aeiou" } > 4 ->
            "The Master of Vowels"
        else -> "The Renowned Hero"
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }

    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        name = newName
    }

}
```

Now, take a look at the resulting decompiled bytecode. (Open the decompiler by selecting `Tools → Kotlin → Show Kotlin Bytecode`, then click the `Decompile` button.) IntelliJ will show you the fully decompiled implementation of `Player.kt`. We have tidied up this output to emphasize the relevant details below:

```
...
public final class Player {
    @NotNull
    private String name = "madrigal";

    @NotNull
    public final String getName() {
        // Returns the value of this.name
    }

    private final void setName(String value) {
        // Assigns the value of this.name to value.trim()
    }

    @NotNull
    public final String getTitle() {
        // Evaluates your when expression to generate player titles
    }

    public final void castFireball(int numFireballs) {
        ...
    }

    // $FF: synthetic method
    public static void castFireball$default(...) {
        ...
    }

    public final void changeName(@NotNull String newName) {
        // Calls setName(newName) and narrates your message
    }
}
}
```

From top to bottom, this class has:

- a field for `name` plus a getter and a setter (called **`getName`** and **`setName`**)
- a **`getTitle`** getter for `title`
- the **`castFireball`** function
- a “synthetic” function used to handle **`castFireball`**’s default argument
- the **`changeName`** function

`name` has a field, getter, and setter – because it is a `var` – while `title` only has a getter, because it is a computed `val`.

Now try changing the `name` property from a `var` to a `val`. (Comment out the shaded code, instead of deleting it, because you will need to undo this change before proceeding with the next chapter.)

Listing 13.15 Changing the var to a val (Player.kt)

```
...
class Player {

    var val name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    /* private set(value) {
        field = value.trim()
    */
    ...
}
```

```

    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        // name = newName
    }
}

```

And observe the resulting decompiled bytecode by launching the decompiler again. (The strike-through here is to emphasize what is missing.)

```

public final class Player {
    @NotNull
    private String name = "madrigal";

    @NotNull
    public final String getName() {
        // Returns the value of this.name
    }

    private final void setName(String value) {
        // Assigns the value of this.name to value.trim()
    }
    ...
}

```

The difference between using the `var` keyword and `val` keyword for the property is the absence of a setter.

You also learned in this chapter that you can define a custom getter or setter for a property. Your getters’ and setters’ implementations appeared directly inside the **getName**, **setName**, and **getTitle** functions.

Computed properties work the same way. Your `title` property only has a getter. The compiler was able to determine that no field was required, since you do not reference the field in the getter.

This particular feature of properties – computing a value, rather than reading a field’s state – is another reason we use the term “read-only” rather than “immutable.” Consider a **Dice** class:

```

class Dice {
    val rolledValue
    get() = (1..6).random()
}

```

The result of reading **Dice**’s `rolledValue` property is a random value ranging from 1 to 6, determined each time the property is accessed – hardly the definition of “immutable.”

When you are done exploring the bytecode, return `Player.kt` to its original state by restoring your `name`’s `var` keyword and setter.

Listing 13.16 Restoring Player (Player.kt)

```

...
class Player {

    val var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    private set(value) {
        field = value.trim()
    } set

    ...
    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        name = newName
    }
}

```


For the More Curious: Guarding Against Mutability

When a property is both nullable and mutable, you must ensure that it is non-null before using its value. For example, players in NyetHack might carry weapons, but a player's weapon could also be null (if they do not have one yet or they have been disarmed). Consider the code below, which prints the name of a player's weapon, if they are wielding one:

```
class Weapon(val name: String)

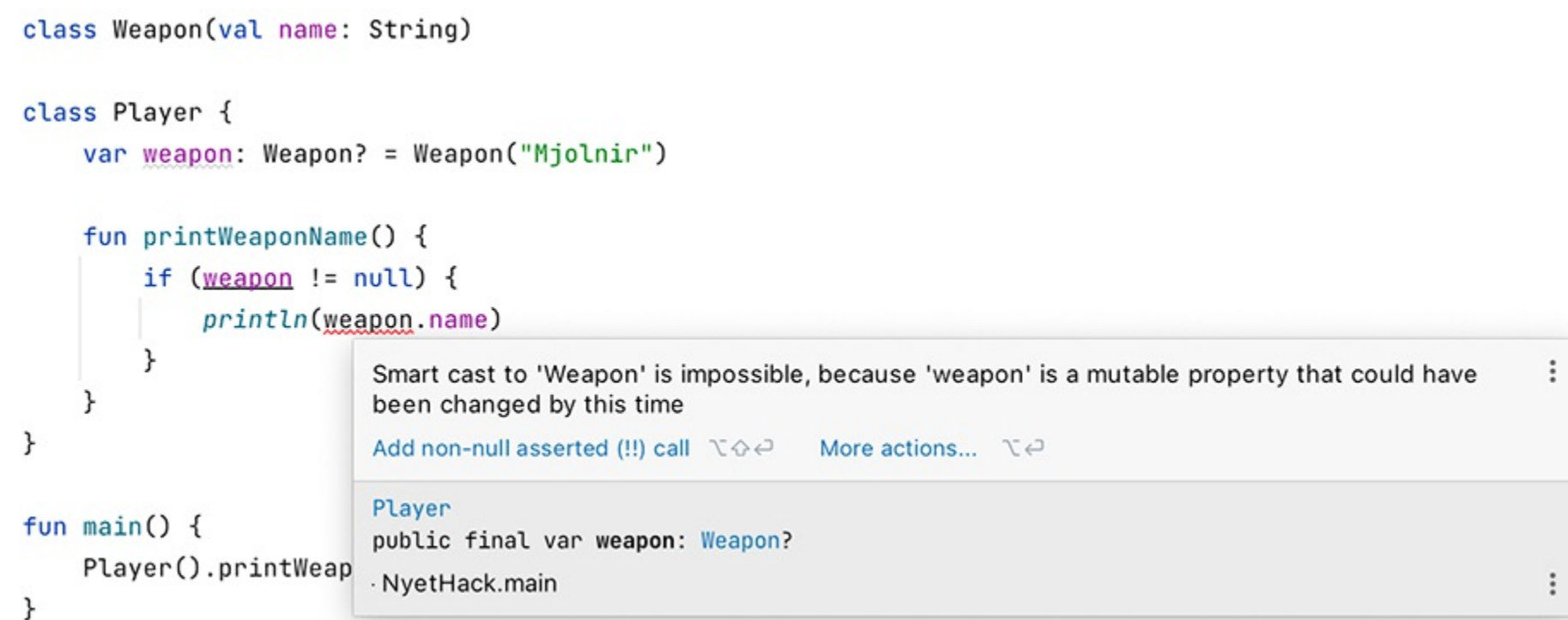
class Player {
    var weapon: Weapon? = Weapon("Mjolnir")

    fun printWeaponName() {
        if (weapon != null) {
            println(weapon.name)
        }
    }
}

fun main() {
    Player().printWeaponName()
}
```

You may be surprised to learn that this code does not compile. Check out the error to see why ([Figure 13.4](#)):

Figure 13.4 Smart cast to 'Weapon' is impossible



The compiler prevents the code from compiling because of the possibility of what is known as a [race condition](#), which you will learn more about in [Chapter 20](#). A race condition occurs when some other part of your program simultaneously modifies the state of your code in a manner that leads to unpredictable results.

Here, the compiler sees that although `weapon` is checked for a null value, there is still a possibility of the **Player**'s `weapon` property being replaced with a null value between the time that check passed and the time the name of the weapon is printed.

Therefore, unlike the cases you saw in [Chapter 7](#), `weapon` cannot be smart cast within the null check. The earlier examples involved variables declared within a function; since only one call to that function has access to such variables at a time, smart casting is permitted. But `weapon` is a mutable class property, so it could be changed while your function is running.

The compiler balks at the code above because it cannot safely say that `weapon` will not be null. (You can, however, use smart casting on `val` properties as long as the property is defined in your code, is not a computed property, and cannot be modified by other classes.)

One way to fix this problem is to use a scope function like **let**, which you read about in [Chapter 12](#), to guard against null:

```
class Player {
    var weapon: Weapon? = Weapon("Mjolnir")

    fun printWeaponName() {
        weapon?.let {
            println(it.name)
        }
    }
}
```

This code compiles, thanks to the **let** scope function. Instead of referring to the class property, this code also uses `it`, the argument to **let**, which is a local variable that exists only within the scope of the anonymous function. Therefore, the `it` variable is guaranteed to not be changed by another part of the program.

The smart cast issue is avoided entirely, because instead of dealing with the original nullable property, this code

uses a read-only, non-nullable local variable (because **let** is called after the safe call operator: `weapon?.let`).

For the More Curious: Package Private

In this chapter, we discussed public and private visibility levels. As you learned, a Kotlin class, function, or property is public by default (without a visibility modifier), which means it is usable by any other class, function, or property in the project.

If you are familiar with Java, you may have noticed that the default access level is different between the two languages: By default, Java uses package private visibility, which means that methods, fields, and classes with no visibility modifier are usable from classes belonging to the same package only.

And package private visibility is not only not the default in Kotlin – it is not supported at all. Kotlin opted out of supporting package private visibility because it accomplishes little. In practice, it is easily circumvented by creating a matching package and adding a class to it.

On the other hand, Kotlin provides the internal visibility level, which Java does not. Internal visibility marks a function, class, or property as accessible to other functions, classes, and properties within the same [module](#). A module is a discrete unit of functionality that can be run, tested, and debugged independently.

Modules include such things as source code, build scripts, unit tests, deployment descriptors, and so on. `NyetHack` (not the `NyetHack.kt` file but the top-level `NyetHack`) is one module within your project, and an IntelliJ project can contain multiple modules. Modules can also depend on other modules for source files and resources.

Internal visibility is useful for sharing classes within a module while disallowing access from other modules, which makes it a great choice for building libraries in Kotlin.

Initialization

In the last chapter, you saw how to define classes that represent real-world objects. In NyetHack, a player is defined in part by its properties and by its behavior. For all the complexity that can be represented using class properties and functions, you have seen very little so far of how instances of classes come to be.

Think back to how **Player** was defined in the last chapter.

```
class Player {  
    ...  
}
```

Player's class header is quite simple, so instantiating **Player** was also simple:

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

Recall that when you call a class's constructor, an instance of that class is created – a process known as instantiation. This chapter covers the ways classes and their properties can be [initialized](#). When you initialize a variable, property, or class instance, you assign it an initial value to make it ready for use. You will see more constructors, learn about property initialization, and even learn how to bend the rules with late and lazy initialization.

A note about terminology: Technically, an object is *instantiated* when memory is allocated for it, and it is *initialized* when it is assigned a value. However, in practice the terms are often used slightly differently. Often, *initialization* is used to mean “everything required to make a variable, property, or class instance ready to use,” while *instantiation* tends to be limited to “creating an instance of a class.” In this book, we follow this more typical usage.

Constructors

Player now contains behavior and data you defined. For example, you specified a `name` property:

```
var name = "madrigal"
get() = field.replaceFirstChar { it.uppercaseChar() }
private set(value) {
    field = value.trim()
}
```

With this current implementation, every player begins as “madrigal” and needs to change their name after the fact, which might involve roping in various bureaucracies in NyetHack. It would be better to create the **Player** instance with the correct name.

This is where a [primary constructor](#) comes into play. A constructor allows its caller to specify the initial values that an instance of a class requires in order to be constructed. Those values are then available for assignment to the properties defined within the class.

Primary constructors

Like a function, a constructor defines expected parameters that must be provided as arguments. To specify what is needed for a **Player** instance to work correctly, you are going to define the primary constructor in **Player**’s header. Update `Player.kt` to accept the player’s name through a primary constructor. Also, request a bit of additional information about the player, which will become useful as you build out new features in NyetHack.

Listing 14.1 Defining a primary constructor (Player.kt)

```
package com.bignerdranch.nyethack

class Player(
    initialName: String,
    hometown: String,
    healthPoints: Int,
    isImmortal: Boolean
) {
    var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercaseChar() }
    private set(value) {
        field = value.trim()
    }

    val hometown = hometown

    var healthPoints = healthPoints

    val isImmortal = isImmortal

    val title: String
    get() = ...
    ...
}
```

Now, to create an instance of **Player**, you will provide arguments that match the parameters you added to the constructor. Instead of hardcoding the value for the player’s `name` property, for example, you pass an argument to **Player**’s primary constructor.

Notice that the names of the constructor parameters match or closely match the names of the properties on the object. This is conventional, but not required. If you want to make clear whether you are referring to the constructor parameter or the property, you can give them different names. If a property and constructor parameter are named the same, Kotlin will use the constructor parameter for all initialization steps of the class and the property elsewhere.

If you ever want to confirm that you are referencing the correct property or variable, you can place your text caret over the symbol in question and press Command-B (Ctrl-B) to jump to its definition. You can also Command-click (Ctrl-click) the variable name. If Kotlin is reading from the constructor argument, IntelliJ will take you to the parameter. Otherwise, if it is reading from a property, it will take you to the property declaration.

This trick works for navigating to function and class definitions as well, and it is often helpful for navigating through your project or seeing how a certain class or function is implemented.

To use your new constructor, change the call to **Player**’s constructor in **main** to include the new information. Also, remove the code to change the player’s name when NyetHack is started, and add information about the player’s vitals to the narration.

(We have broken the first narration string onto two lines to fit the printed page; you should enter it on one line.)

Listing 14.2 Calling the primary constructor (NyetHack.kt)

```
package com.bignerdranch.nyethack

val player = Player("Jason", "Jacksonville", 100, false)

fun main() {
    narrate("${player.name} is ${player.title}")
    player.changeName("Aurelia")
    // changeNarratorMood()
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
}
```

```

narrate("${player.name} of ${player.hometown}, ${player.title},
        heads to the town square")
narrate("${player.name}, $mortality, has ${player.healthPoints} health points")

visitTavern()
player.castFireball()
}
...

```

Consider how much functionality the primary constructor has added to **Player**: Without it, players would always start out as Madrigal and would not be able to change read-only values like their hometown and immortality status. Now, a player can be named anything and can be created with any hometown, any number of health points, and either immortality status – none of **Player**’s data is hardcoded.

Run NyetHack and confirm that your output starts out like this:

```

Jason of Jacksonville, The Renowned Hero, heads to the town square
Jason, a mortal, has 100 health points
Jason enters Taernyl's Folly
...

```

Defining properties in a primary constructor

Notice the one-to-one relationship between the constructor parameters in **Player** and the class properties: You have a parameter and a class property for each property to be specified when the player is constructed.

For properties that use the default getter and setter, Kotlin allows you to specify both in one definition, rather than having to assign them using temporary variables. `name` uses a custom getter and setter, so it cannot take advantage of this feature, but **Player**’s other properties can.

Update the **Player** class to define `hometown`, `healthPoints`, and `isImmortal` as properties in **Player**’s primary constructor.

Listing 14.3 Defining properties in the primary constructor (Player.kt)

```

class Player(
    initialName: String,
    val hometown: String,
    var healthPoints: Int,
    val isImmortal: Boolean
) {

    var name = initialName
    get() = field.replaceFirstChar { it.uppercaseChar() }
    private set(value) {
        field = value.trim()
    }

    val hometown = hometown

    var healthPoints = healthPoints

    val isImmortal = isImmortal
    ...
}

```

For each constructor parameter, you specify whether it is mutable or read-only. By specifying the parameters with `val` or `var` keywords in the constructor, you define properties for the class, their mutability, and the parameters the constructor will expect arguments for. You also implicitly assign each property to the value passed to it as an argument.

Duplication of code makes it harder to make changes. Generally, we prefer this way of defining class properties because it leads to less duplication. You cannot use this syntax for `name`, because of its custom getter and setter, but in other cases defining a property in a primary constructor is often the most straightforward choice.

Secondary constructors

Constructors come in two flavors: primary and secondary. The constructor you have just defined is a primary constructor. When you specify a primary constructor, you say, “These parameters are required for any instance of this class.” You can also specify a secondary constructor to provide alternative ways to construct the class while still meeting the requirements of the primary constructor.

A secondary constructor must either call the primary constructor, providing it all the arguments it requires, or call through to another secondary constructor – which follows the same rule. For example, say you know that in most cases a player will begin with 100 health points and will be mortal. You can define a secondary constructor to provide that configuration. Add a secondary constructor to **Player**:

Listing 14.4 Defining a secondary constructor (Player.kt)

```

class Player(
    initialName: String,
    val hometown: String,
    var healthPoints: Int,
    val isImmortal: Boolean
) {
    ...
    val title: String
    get() = when {
        ...
    }
}

```

```

        constructor(name: String, hometown: String) : this(
            initialName = name,
            hometown = hometown,
            healthPoints = 100,
            isImmortal = false
        )
    }
}

```

You can define multiple secondary constructors for different combinations of parameters. This secondary constructor calls through to the primary constructor with a certain set of parameters. The `this` keyword in this case refers to the instance of the class for which this constructor is defined. Specifically, this is calling into another constructor defined in the class – the primary constructor.

Because this secondary constructor provides default values for `healthPoints` and `isImmortal`, you do not need to pass arguments for those parameters when calling it. Call **Player**'s secondary constructor from `NyetHack.kt` instead of its primary constructor.

Listing 14.5 Calling a secondary constructor (NyetHack.kt)

```

...
val player = Player("Jason", "Jacksonville", 100, false)

fun main() {
    ...
}
...

```

You can also use a secondary constructor to define initialization logic – code that will run when your class is instantiated. For example, add an expression that rewards players named Jason with a much larger health pool.

Listing 14.6 Adding logic to a secondary constructor (Player.kt)

```

class Player(
    initialName: String,
    val hometown: String,
    var healthPoints: Int,
    val isImmortal: Boolean
) {
    ...
    constructor(name: String, hometown: String) : this(
        initialName = name,
        hometown = hometown,
        healthPoints = 100,
        isImmortal = false
    ) {
        if (name.equals("Jason", ignoreCase = true)) {
            healthPoints = 500
        }
    }
    ...
}

```

Secondary constructors are useful for defining alternative logic to be run on instantiation. A secondary constructor's logic only applies when called – it has no effect on the other constructors for a class. For this reason, secondary constructors cannot be used to define properties like primary constructors can. Class properties must be defined in the primary constructor or at the class level.

Run `NyetHack` to see that Jason is not immortal and has many health points, showing that **Player**'s secondary constructor was called from `NyetHack.kt`.

Default arguments

When defining a constructor, you can also specify default values that should be assigned if an argument is not provided for a specific parameter. You have seen these default arguments in the context of functions, and they work the same way with both primary and secondary constructors. For example, set the default value for `hometown` with a default parameter value of "Neversummer" in the primary constructor.

Listing 14.7 Defining a default argument in a constructor (Player.kt)

```

class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    var healthPoints: Int,
    val isImmortal: Boolean
) {
    ...
    constructor(name: String, hometown: String) : this(
        initialName = name,
        hometown = hometown,
        healthPoints = 100,
        isImmortal = false
    ) {
        if (name.equals("Jason", ignoreCase = true)) {
            healthPoints = 500
        }
    }
    ...
}

```

Because you removed an argument from your secondary constructor, you will also need to update your definition of `player` accordingly.

Listing 14.8 Using the default argument (NyetHack.kt)

```

...
val player = Player("Jason", "Jacksonville")

fun main() {
    ...
}

```



```
}  
...
```

Run your code and confirm that while Jason may be a die-hard Jaguars fan, he is now from Neversummer instead of Jacksonville. Using default arguments and secondary constructors, there are many ways to define the combinations of arguments you want to allow for a constructor. For your **Player** class, your options now look like this:

<code>Player("Jason", "Jacksonville", 40, true)</code>	Primary constructor with all parameters specified
<code>Player("Madrigan", healthPoints = 40, isImmortal = false)</code>	Primary constructor with Neversummer hometown via default argument and other parameters specified
<code>Player("Estragon")</code>	Secondary constructor with name specified and Neversummer, 100 HP, and mortal status via default arguments

Named arguments

The more default arguments you use, the more options you have for calling your constructor. More options can open the door for more ambiguity, so Kotlin provides named constructor arguments, just like the named arguments that you have used to call functions. The rules for how named arguments work are the same for regular functions and constructors.

Named argument syntax lets you include the parameter name for each argument to improve readability. Compare the following two options for constructing an instance of **Player**:

```
val player = Player(  
    initialName = "Madrigan",  
    hometown = "Neversummer",  
    healthPoints = 40,  
    isImmortal = true  
)  
  
val player = Player("Madrigan", "Neversummer", 40, true)
```

Which option do you find to be more readable? If you chose the first, we agree with your judgment.

Named argument syntax is especially useful when you have multiple parameters of the same type. For your **Player** class, named parameters make it clear whether a player is named Madison and comes from Austin or is named Austin and comes from Madison.

This reduced ambiguity leads to another benefit: Named arguments allow you to specify the arguments to a function or constructor in any order. If parameters are unnamed, then you must be careful to exactly match the order they are defined in.

You may have noticed that the secondary constructor you wrote for **Player** uses named arguments, similar to the ones that you saw in [Chapter 4](#):

```
class Player(  
    initialName: String,  
    val hometown: String = "Neversummer",  
    var healthPoints: Int,  
    val isImmortal: Boolean  
) {  
    ...  
    constructor(name: String, hometown: String) : this(  
        initialName = name,  
        healthPoints = 100,  
        isImmortal = false  
    ) { ...  
    }  
    ...  
}
```

When you have more than a few arguments to provide to a constructor or function, we recommend using named parameters. They make it easier for readers to keep track of which argument is being passed as which parameter.

Initializer Blocks

In addition to the primary and secondary constructors, you can also specify an [initializer block](#) for a class in Kotlin. The initializer block is a way to set up variables or values as well as perform validation – like checking to make sure that the arguments to the constructor are valid. The code it holds is executed when the class is constructed.

For example, players have certain requirements as they are constructed: A player must begin the game with at least one health point. Their name must not be blank.

Use an initializer block, denoted by the `init` keyword, to enforce these requirements with preconditions.

Listing 14.9 Defining an initializer block (`Player.kt`)

```
class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    var healthPoints: Int,
    val isImmortal: Boolean
) {
    ...
    val title: String
    get() = when {
        ...
    }

    init {
        require(healthPoints > 0) { "healthPoints must be greater than zero" }
        require(name.isNotBlank()) { "Player must have a name" }
    }

    constructor(name: String) : this(
        initialName = name,
        healthPoints = 100,
        isImmortal = false
    ) { ...
    }
    ...
}
```

The code in the initializer block will be called when the class is instantiated – no matter which constructor for the class is called, primary or secondary. If either of these preconditions fails, then an **`IllegalArgumentException`** is thrown. (You can test this by passing **`Player`** different parameters in the Kotlin REPL.)

These requirements would be difficult to encapsulate in a constructor or a property declaration. If you want, you can also place property assignments inside initializer blocks. This is typically only done when you cannot compute the initial value with a single expression, and it is a nice tool to keep in your back pocket.

You can also use initializer blocks to assign properties if you want to separate the declaration and assignment. This is particularly useful if you need to define complex logic involving several statements to compute a property's initial value. For example, if you wanted to calculate a player's initial inventory, you might use an `init` block that looks like this:

```
class Player {
    ...
    val inventory: List<String>

    init {
        val baseInventory = listOf("waterskin", "torches")
        val classInventory = when (playerClass) {
            "archer" -> listOf("arrows")
            "wizard" -> listOf("arcane staff", "spellbook")
            "rogue" -> listOf("lockpicks", "crowbar")
            else -> emptyList()
        }
        inventory = baseInventory + classInventory
    }
}
```

Initialization Order

You have seen how to initialize properties or add logic to the initialization of properties in various ways – [inline](#) in a primary constructor, initialized at declaration, initialized in a secondary constructor, or initialized in an initializer block. It is possible for the same property to be referenced in multiple initializers, so the order in which they are executed is important.

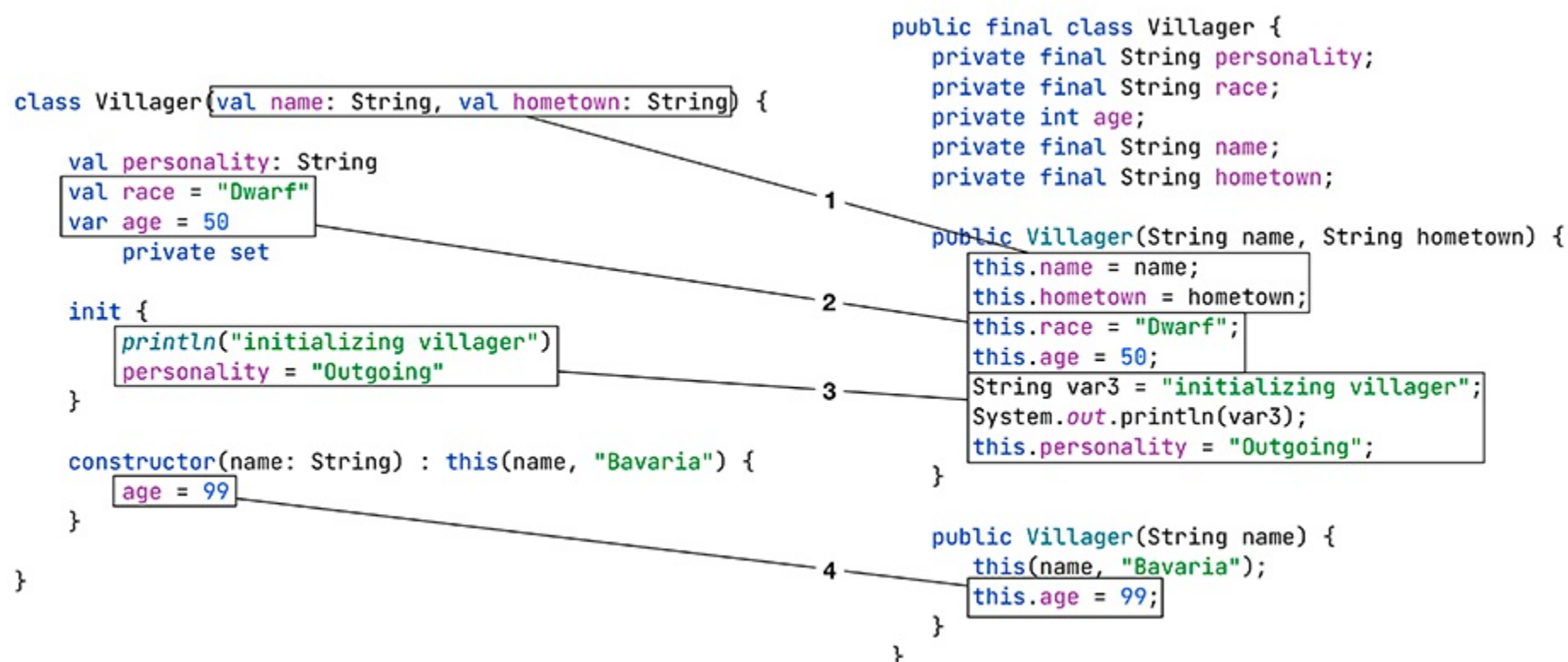
Consider the following **Villager** class, which represents the residents of Kronstadt.

```
class Villager(val name: String, val hometown: String) {  
    val personality: String  
    val race = "Dwarf"  
    var age = 50  
    private set  
  
    init {  
        println("initializing villager")  
        personality = "Outgoing"  
    }  
  
    constructor(name: String) : this(name, "Bavaria") {  
        age = 99  
    }  
}
```

Suppose you construct an instance with the class's secondary constructor by calling `Villager("Estragon")`. What order are these expressions executed in?

To find out, it is helpful to examine the resulting field initialization order and method invocations in the decompiled Java bytecode. [Figure 14.1](#) shows this **Villager** class on the left. The abbreviated decompiled Java bytecode on the right shows the resulting initialization order:

Figure 14.1 Initialization order for the Villager class (decompiled bytecode)



The resulting initialization order is as follows:

- properties declared in the primary constructor (1: `name` and `hometown`)
- `init` blocks and property assignments, in the order they are declared (2: the `race` and `age` properties; 3: the call to `println` and the assignment of `personality`)
- initializers for secondary constructors (4: the `age = 99` assignment)

(The initialization order of `init` blocks and property assignments depends on the order they are specified in. If the `init` block were defined before the `race` and `age` properties are set, it would be executed before those properties are assigned.)

Delaying Initialization

Wherever it is declared, a class property must be initialized when the class instance is constructed. This rule is an important part of Kotlin’s null safety system, because it means that all non-nullable properties of a class are initialized with a non-null value when the constructor for that class is called. When you instantiate an object, you can immediately reference any property on that object, from within or outside of the class.

Despite its importance, you can bend this rule. Why would you? Because you do not always have control over how or when a constructor is called – which often happens with frameworks and libraries like Android, Spring, and JUnit.

Late initialization

Sometimes, you will find yourself in a situation where you cannot initialize a property when your object is being initialized and must wait until later to obtain a property’s value. This may be related to your framework (for example, Android apps have several components that are initialized in a function called **onCreate** instead of in a constructor), or to the design of your application.

Traditionally, the solution to this issue is to assign the property to null for the purposes of initialization and then set the property to its actual value when possible. This works, but has an unfortunate downside: Every time you access that property, you must check whether it is null.

In some cases, it makes sense to have a nullable value. But in other situations, you will have a property that, once initialized, always has a value – and then Kotlin’s null safety requirements quickly become an annoyance.

This is where [late initialization](#) becomes important. Late initialization lets you bend Kotlin’s rules on initialization just a bit.

Any var property declaration can be appended with the `lateinit` keyword. This tells the Kotlin compiler to let you skip initializing the property when the class is instantiated.

```
class Arena {
    var isArenaOpen = false
    lateinit var opponentName: String

    fun prepareArena() {
        isArenaOpen = true
        opponentName = getWillingCombatants().random()
    }

    private fun getWillingCombatants() =
        listOf("Cornelius", "Cheryl", "Ralph", "Deborah")
}
```

This is useful – but must be treated with care. Not every variable deserves to be marked as `lateinit`. Properties on your **Player** class, for example, should likely not be declared as `lateinit`, because all information about a player can be obtained before creating an instance of the class.

Still, sometimes late initialization cannot be avoided. Take a look at the `player` variable in `NyetHack.kt`:

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

fun main() {
    // changeNarratorMood()
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name} of ${player.hometown}, ${player.title},
        heads to the town square")
    narrate("${player.name}, $mortality, has ${player.healthPoints}
        health points")

    visitTavern()
    player.castFireball()
}
...
```

The `lateinit` keyword can also be used on top-level properties, and `player` is a good candidate for late initialization. You may be wondering why we recommend this: Should we not create a player immediately when starting `NyetHack`?

Remember your logic to prompt the user for a name in **promptHeroName**? You have not used this function since you introduced the **Player** class, but eventually you will want to restore it.

Prompting for the name will need to happen before creating the player, which creates a predicament: How can you initialize `player` in **main** but keep it as a top-level property? This is where late initialization shines.

Mark `player` as a `lateinit var` and restore your function call to ask the player for their name. (But keep the details in **promptHeroName** commented out, because you still have much to build in `NyetHack` and will not want to enter a name each time you test your code.)

Listing 14.10 Using `lateinit` (`NyetHack.kt`)

```
val player = Player("Jason")
lateinit var player: Player
```

```

fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name} of ${player.hometown}, ${player.title},
        heads to the town square")
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")

    visitTavern()
    player.castFireball()
}

private fun promptHeroName(): String {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Prints the message in yellow
        "\u001b[33;1m$message\u001b[0m"
    }

    /*val input = readLine()
    require(input != null && input.isNotEmpty()) {
        "The hero must have a name."
    }

    return input*/
    println("Madrigal")
    return "Madrigal"
}

```

Run NyetHack and confirm that your new initialization logic is working as expected:

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Madrigal of Neversummer, The Renowned Hero, heads to the town square
...

```

When you use `lateinit`, you tell the compiler, “I promise that I will set this variable before I use it.” As long as you initialize your `lateinit` variable before it is accessed, then there is no problem. To see what happens if you fail to hold up your end of the bargain, try to access `player` before assigning it.

Listing 14.11 Breaking promises with the compiler (NyetHack.kt)

```

lateinit var player: Player

fun main() {
    narrate("Welcome to NyetHack, ${player.name}!")
    val playerName = promptHeroName()
    player = Player(playerName)
    ...
}
...

```

The compiler does not complain about this code. But when you run NyetHack, it will crash with an **UninitializedPropertyAccessException**, and you will see the following output:

```

Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit
property player has not been initialized
    at com.bignerdranch.nyethack.NyetHackKt.getPlayer(NyetHack.kt:3)
    at com.bignerdranch.nyethack.NyetHackKt.main(NyetHack.kt:6)
    at com.bignerdranch.nyethack.NyetHackKt.main(NyetHack.kt)

```

Any time you attempt to read a `lateinit` property before it has been assigned, your code will crash in this way. If you need to, you can check whether a late-initialized variable has been initialized, like this:

```

lateinit var player: Player

fun main() {
    if (::player.isInitialized) {
        narrate("Welcome to NyetHack, ${player.name}!")
    }
    val playerName = promptHeroName()
    player = Player(playerName)
    ...
}

```

You can check **isInitialized** when there is any uncertainty about whether a `lateinit` variable is initialized to avoid an **UninitializedPropertyAccessException**. However, **isInitialized** should be used sparingly – it should not be added to every `lateinit`, for example. If you are using **isInitialized** a lot, it is likely an indicator that you should be using a nullable type instead.

Fix the crash by undoing your last change before moving on.

Listing 14.12 Upholding promises with the compiler (NyetHack.kt)

```

lateinit var player: Player

fun main() {
    narrate("Welcome to NyetHack, ${player.name}!")
    val playerName = promptHeroName()
    player = Player(playerName)
    ...
}
...

```

Although `lateinit` is sometimes required, it is not without its limitations. `lateinit` can only be used with `var` properties, because there is no way to guarantee that the property can only be set once, and you cannot define a custom getter or setter for any `lateinit` vars. `lateinit` also cannot be used if the property’s type is **Boolean**, **Char**, or any numeric type (including **Int**, **Double**, and **UInt**). Behind the scenes, a `lateinit` var is implemented with null values, and it is not possible for Kotlin to assign these primitive types to null at runtime.

Lazy initialization

Late initialization is not the only way to delay initialization. You can also hold off on initializing a variable until it

is accessed for the first time. This concept is known as [lazy initialization](#), and, despite the name, it can actually make your code more efficient.

Most of the properties that you have initialized in this chapter have been pretty lightweight – single objects, like a **String**, that could be calculated almost instantly. But some properties will hold more complex values. They may require the instantiation of multiple objects, or they may involve some more computationally intensive task when being initialized, like reading from a file.

If your property triggers a large number tasks like these, or if your class does not require access to a property right away, then lazy initialization could be a good choice.

Suppose that players could be given a prophecy describing heroic actions they have yet to perform. Obtaining this prophecy involves locating a suitable fortune teller, which can be difficult. Not every player needs to have their prophecy foretold, because it will happen whether it is told to them or not. Prophecies are also irrevocable: Once a player has been given a prophecy they cannot have it changed.

Lazy initialization implements exactly this pattern. If a prophecy is computed lazily, the player will not venture out to obtain the prophecy until they need it for the first time. After the player has obtained a prophecy, they will remember it and will instantly return it the next time you ask for it.

To see these concepts in action, introduce a new prophecy property on **Player** as well as a new function called **propheSize**.

Listing 14.13 Lazily obtaining prophecies (Player.kt)

```
class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    var healthPoints: Int,
    val isImmortal: Boolean
) {
    ...
    val title: String
    get() = when {
        ...
    }

    val prophecy by lazy {
        narrate("$name embarks on an arduous quest to locate a fortune teller")
        Thread.sleep(3000)
        narrate("The fortune teller bestows a prophecy upon $name")

        "An intrepid hero from $hometown shall some day " + listOf(
            "form an unlikely bond between two warring factions",
            "take possession of an otherworldly blade",
            "bring the gift of creation back to the world",
            "best the world-eater"
        ).random()
    }
    ...
    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        name = newName
    }

    fun propheSize() {
        narrate("$name thinks about their future")
        narrate("A fortune teller told Madrigal, \"$prophecy\"")
    }
}
```

Take a look at the new `by lazy` syntax. The `by` keyword indicates that you are implementing this property using a property delegate, which you can learn more about in [the section called For the More Curious: Property Delegates](#) near the end of this chapter. The **lazy** function is a delegate that defines the lazy initialization behavior.

prophecy remains uninitialized until it is referenced for the first time. At that point, all the code in **lazy**'s lambda is executed. Importantly, this code is only executed once – the first time that the delegated property (prophecy, here) is accessed by **propheSize**. Future access to the lazy property will use a cached result – the string returned by the lambda – instead of performing the expensive computation again.

Inside the lambda expression, you called `Thread.sleep(3000)`. This function causes your code to pause for 3,000 milliseconds (that is, 3 seconds) before resuming execution. Although you do not need to add an import statement for it, bear in mind that **Thread** is a Java class.

Now, to see lazy initialization in action, insert two calls to **propheSize** in **main**:

Listing 14.14 Using a lazy property (NyetHack.kt)

```
...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    player.propheSize()
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name} of ${player.hometown}, ${player.title},
        heads to the town square")
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")

    visitTavern()
    player.castFireball()
    player.propheSize()
}
...
```

When you run NyetHack, watch your output. It will look like the following – but notice that your program pauses

for 3 seconds only one time, despite reading from prophecy twice.

```
Welcome to NyetHack!  
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
Madrigal thinks about their future  
Madrigal embarks on an arduous quest to locate a fortune teller  
The fortune teller bestows a prophecy upon Madrigal  
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some  
    day take possession of an otherworldly blade"  
Madrigal of Neversummer, The Renowned Hero, heads to the town square  
...  
A glass of Fireball springs into existence (x2)  
Madrigal thinks about their future  
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some  
    day take possession of an otherworldly blade"
```

Lazy initialization is useful, but it has a bit of overhead – both in terms of extra syntax and in terms of performance – so stick to using it for more computationally needy tasks. For properties that can be computed trivially, you can harm your program’s performance by unnecessarily using the lazy delegate. But for properties that are very expensive to initialize, using lazy initialization can be a great way to defer work until it is needed, which can help your program feel more responsive.

And with that, you have seen what there is to see when it comes to initializing an object in Kotlin. Most often, your experience will be quite straightforward: You call a constructor, and you get a reference to an instance of a class to do with what you will. That said, you have other options when initializing an object in Kotlin, and understanding those options can help you write clean, efficient code.

In the next chapter you will learn about inheritance, an object-oriented principle that allows you to share data and behavior between related classes.

For the More Curious: Initialization Gotchas

You saw earlier in the chapter that order is important when using initializer blocks – you must ensure that all properties used in the block are initialized before the initializer block is defined. Take a look at the following code, which shows this initializer block ordering problem:

```
class Player() {
    init {
        val healthBonus = health.times(3)
    }

    val health = 100
}

fun main() {
    Player()
}
```

This code would not compile, because the `health` property is not initialized at the point that it is used by the `init` block. As we mentioned earlier, when a property is used within an `init` block, the property initialization must happen before it is accessed. When `health` is defined before the initializer block, the code compiles:

```
class Player() {
    val health = 100

    init {
        val healthBonus = health.times(3)
    }
}

fun main() {
    Player()
}
```

There are a couple of similar, but more subtle, scenarios that trip up unwary programmers. For example, in the following code, a `name` property is declared, then a **firstLetter** function reads the first character from the property:

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

    private fun firstLetter() = name[0]

    init {
        println(firstLetter())
        name = "Madrigal"
    }
}

fun main() {
    Player()
}
```

This code will compile, because the compiler sees that the `name` property is initialized in the `init` block, a legal place to assign an initial value. But running this code would result in a runtime error (a **NullPointerException** on the JVM, **TypeError** in JS, and a segmentation fault on native platforms), because the **firstLetter** function (which uses the `name` property) is called before the `name` property is assigned an initial value in the `init` block.

The compiler does not inspect the order properties are initialized in compared to the functions that use them within the `init` block. When defining an `init` block that calls functions that access properties, it is up to you to ensure that you have initialized those properties before calling the functions. When `name` is assigned before **firstLetter** is called, the code compiles *and* will run without error:

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

    private fun firstLetter() = name[0]

    init {
        name = "Madrigal"
        println(firstLetter())
    }
}

fun main() {
    Player()
}
```

One more tricky scenario is shown in the following code, in which two properties are initialized:

```
class Player(name: String) {
    val playerName: String = initPlayerName()
    val name: String = name

    private fun initPlayerName() = name
}

fun main() {
    println(Player("Madrigal").playerName)
}
```

Again, this code compiles, because the compiler sees that all properties have been initialized. But running the code would result in the unsatisfying output `null`.

What is the problem here? When `playerName` is initialized with the **initPlayerName** function, the compiler assumes that `name` is initialized – but when **initPlayerName** is called, `name` is actually not yet initialized.

In this case, once again, order matters. The initialization order of the two properties must be reversed. With that done, the **Player** class compiles and returns a non-null `name` value:


```
class Player(name: String) {  
    val name: String = name  
    val playerName: String = initPlayerName()  
  
    private fun initPlayerName() = name  
}  
  
fun main() {  
    println(Player("Madrigal").playerName)  
}
```

For the More Curious: Property Delegates

Lazy initialization is implemented in Kotlin using a mechanism known as a [delegate](#). Delegates define templates for how a property behaves.

You use a delegate with the `by` keyword. The Kotlin standard library includes some delegates that are already implemented for you: **lazy**, **observable**, **vetoable**, and **notNull**.

When you use the `by` keyword, you tell Kotlin to use the **get** and **set** implementation provided by the delegate you are using. This also means that you cannot define a custom getter or setter when using any property delegate.

In practice, **lazy** is by far the most commonly used delegate. It is rare to see the other built-in delegates in a production codebase, although some Kotlin-focused libraries like Koin and Jetpack Compose define their own delegates that you will find useful.

It is also possible to define your own delegates if you find yourself frequently writing the same custom getters and setters in your code. To do this, take a look at `ReadOnlyProperty` (if you want to write a delegate for `val` properties) or `ReadWriteProperty` (if you want to write a delegate for both `var` and `val` properties). You will need to implement one of these interfaces, and you will learn more about how to do this in [Chapter 17](#).

Challenge: The Riddle of Excalibur

As you learned in [Chapter 13](#), you can specify your own getter and setter for a property. Now that you have seen how properties and their classes are initialized, we have a riddle for you.

Every great sword has a name. Define a class called **Sword** in the Kotlin REPL that reflects this truth.

Listing 14.15 Defining Sword (REPL)

```
class Sword(name: String) {
    var name = name
    get() = "The Legendary $field"
    set(value) {
        field = value.lowercase().reversed().capitalize()
    }
}
```

What is printed when you instantiate a **Sword** and reference `name`? (Try to answer for yourself before you check the REPL.)

Listing 14.16 Referencing name (REPL)

```
val sword = Sword("Excalibur")
println(sword.name)
```

What is printed when you reassign `name`?

Listing 14.17 Reassigning name (REPL)

```
sword.name = "Gleipnir"
println(sword.name)
```

Finally, add an initializer block to **Sword** that assigns `name`.

Listing 14.18 Adding an initializer block (REPL)

```
class Sword(name: String) {
    var name = name
    get() = "The Legendary $field"
    set(value) {
        field = value.lowercase().reversed().capitalize()
    }

    init {
        this.name = name
    }
}
```

What is printed when you instantiate **Sword** and reference `name` now?

Listing 14.19 Referencing name again (REPL)

```
val sword = Sword("Excalibur")
println(sword.name)
```

This challenge will test your knowledge of both initializers and custom property getters and setters.

Inheritance

[*Inheritance*](#) is an object-oriented principle you can use to define hierarchical relationships between types. In this chapter you will use inheritance to share data and behavior between related classes.

To get a handle on inheritance, consider an example outside of programming. Cars and trucks have much in common: They each have wheels, an engine, etc. They also have some different features. Using inheritance, you could define the things that they have in common in a shared class, **Automobile**, so that you do not have to implement the steering, engine, and so on in both **Car** and **Truck**. **Car** and **Truck** would inherit those shared features, making slight modifications as necessary, and each would then define its own unique features as well.

In NyetHack, you will put inheritance to work by adding a series of rooms so that your player has places to go.

Defining the Room Class

Begin by creating a new class file in the `com.bignerdranch.nyethack` package called `Room.kt`. When you created `Player.kt`, you started with a blank file and added the class declaration yourself. This time, let IntelliJ set up the declaration for you by choosing Class for the file type when you create it:

```
class Room {  
}
```

In the future, you can use either method to set up your class files.

`Room.kt` will contain a new class called **Room** that will represent one square in NyetHack's coordinate plane. Later, you will define a particular kind of room in a class that inherits qualities from **Room**.

To begin, **Room** will have one property – `name` – and two functions, **description** and **enterRoom**. **description** returns a **String** describing the room (for now, it will just be the room's name). **enterRoom** defines the behavior of the room and will print to the console what the player sees or experiences inside the room. These are features you want for every room in NyetHack.

Add the **Room** class definition to `Room.kt`:

Listing 15.1 Declaring the Room class (Room.kt)

```
class Room(val name: String) {  
  
    fun description() = name  
  
    fun enterRoom() {  
        narrate("There is nothing to do here")  
    }  
}
```

Back in `NyetHack.kt`, test your new **Room** class by creating a **Room** instance when the game starts in **main** and printing the result of its **description** function. While you are there, remove the call to **visitTavern** – you will be refactoring the tavern later in this chapter to use the new **Room** class.

Listing 15.2 Printing the room description (NyetHack.kt)

```
...  
fun main() {  
    narrate("Welcome to NyetHack!")  
    val playerName = promptHeroName()  
    player = Player(playerName)  
    // changeNarratorMood()  
    player.propheSize()  
  
    var currentRoom = Room("The Foyer")  
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"  
    narrate("${player.name} of ${player.hometown}, ${player.title},  
        heads-to-the-town-square  
        is in ${currentRoom.description()}")  
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")  
    currentRoom.enterRoom()  
  
    visitTavern()  
    player.castFireball()  
    player.propheSize()  
}  
...
```

Run NyetHack. You should see the following output in the console:

```
Welcome to NyetHack!  
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
Madrigal thinks about their future  
Madrigal embarks on an arduous quest to locate a fortune teller  
The fortune teller bestows a prophecy upon Madrigal  
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some  
    day bring the gift of creation back to the world"  
Madrigal of Neversummer, The Renowned Hero, is in The Foyer  
Madrigal, a mortal, has 100 health points  
There is nothing to do here  
A glass of Fireball springs into existence (x2)  
Madrigal thinks about their future  
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some  
    day bring the gift of creation back to the world"
```

So far, so good ... but kind of boring. Who wants to hang out in a foyer? It is time for Madrigal of Neversummer to go places.

Creating a Subclass

A [subclass](#) shares all properties with the class it inherits from, commonly known as the parent class or [superclass](#).

For example, citizens of NyetHack will need a town square. A town square is a type of **Room** that has the features of a **Room** *plus* special features only town squares will have – like a welcome message that greets entering players. To create the **TownSquare** class, you will subclass **Room**, since they have common features, and then describe how **TownSquare** differs from **Room**.

But before defining a **TownSquare** class, you need to make a change to the **Room** class so that it can be subclassed.

Not every class you write is intended to be part of a hierarchy. By default, classes are closed for extension – meaning they prohibit subclassing. For a class to be subclassed, it must be marked with the open keyword.

Add the open keyword to **Room** so that it can be subclassed.

Listing 15.3 Making the Room class open for subclassing (Room.kt)

```
open class Room(val name: String) {  
    fun description() = name  
    fun enterRoom() {  
        narrate("There is nothing to do here")  
    }  
}
```

Now that **Room** is marked open, create a **TownSquare** class in a new file called `TownSquare.kt`. Make **TownSquare** subclass the **Room** class by using the `:` operator, like so:

Listing 15.4 Declaring the TownSquare class (TownSquare.kt)

```
class TownSquare : Room("The Town Square")
```

The **TownSquare** class declaration includes the class name to the left of the `:` operator and a constructor invocation to the right. The constructor invocation indicates which constructor to call for **TownSquare**'s parent and what arguments to pass to it. In this case, a **TownSquare** is a version of **Room** with the specific name "The Town Square".

But you want more from your town square than just a name. Another way for you to differentiate a subclass from its parent is through [overriding](#). Recall from [Chapter 13](#) that a class uses properties to represent data and functions to represent behavior. Subclasses can override, or provide custom implementations for, both.

Room has two functions, **description** and **enterRoom**. **TownSquare** should provide its own implementation of **enterRoom** to express the joy that comes with your hero entering the town square.

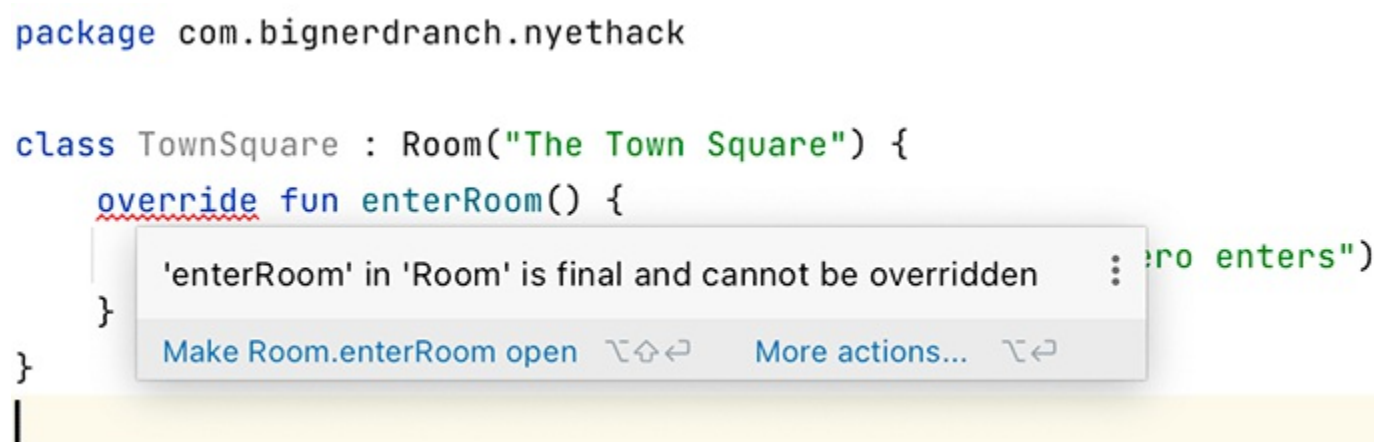
Override **enterRoom** in **TownSquare** using the override keyword:

Listing 15.5 Overriding enterRoom (TownSquare.kt)

```
class TownSquare : Room("The Town Square") {  
    override fun enterRoom() {  
        narrate("The villagers rally and cheer as the hero enters")  
    }  
}
```

When you override **enterRoom**, IntelliJ complains about your override keyword ([Figure 15.1](#)):

Figure 15.1 enterRoom cannot be overridden



IntelliJ is right, as always: There is a problem. In addition to **Room** being marked open, **enterRoom** must also be marked open for you to override it.

Mark the **enterRoom** function in the **Room** class as a function that can be overridden.

Listing 15.6 Making enterRoom an open function (Room.kt)

```
open class Room(val name: String) {  
    fun description() = name  
  
    open fun enterRoom() {  
        narrate("There is nothing to do here")  
    }  
}
```

Now, instead of printing a default statement (“There is nothing to do here”), an instance of **TownSquare** will display the cheering villagers when the hero enters and **enterRoom** is called.

In [Chapter 13](#), you saw how to control the visibility of properties and functions using visibility modifiers. Properties and functions are public by default. You can also make them visible only within the class where they are defined by setting visibility to private.

Protected visibility is a third option that restricts visibility to the class in which a property or function is defined *or* to any subclasses of that class.

In `Room.kt`, add a new protected property called `status` to **Room**.

Listing 15.7 Declaring a protected property (Room.kt)

```
open class Room(val name: String) {  
    protected open val status = "Calm"  
  
    fun description() = "$name (Currently: $status)"  
  
    open fun enterRoom() {  
        narrate("There is nothing to do here")  
    }  
}
```

`status` indicates the general state of the room (other rooms might report a status of dangerous, rowdy, or spooky, for example). It is printed in the description so that the player knows what to expect when entering the room. In the default **Room**, nothing particularly exciting is happening, hence the default value.

Subclasses of **Room** can modify `status` to reflect whether a room is dangerous, but `status` should otherwise be encapsulated to **Room** and its subclasses. This scenario is perfect for the protected keyword: You want to expose a property only to the class where the property is defined and its subclasses.

To override the `status` property in **TownSquare**, you use the override keyword, just as you did with the **enterRoom** function.

Listing 15.8 Overriding status (TownSquare.kt)

```
class TownSquare : Room("The Town Square") {  
    override val status = "Bustling"  
  
    override fun enterRoom() {  
        narrate("The villagers rally and cheer as the hero enters")  
    }  
}
```

Subclasses are not limited to overriding the properties and functions of their superclass. They can also define their own.

NyetHack town squares, for example, are unique among rooms in that they have bells that ring out to announce important happenings. Add a function called **ringBell** and a private variable called `bellSound` to **TownSquare**. `bellSound` holds a string representing the sound that the bell makes, and **ringBell**, called in the **enterRoom** function, returns a string to announce the hero’s entry to the town square.

Listing 15.9 Adding a new property and function to a subclass (TownSquare.kt)

```
class TownSquare : Room("The Town Square") {  
    override val status = "Bustling"  
    private var bellSound = "GWONG"  
  
    override fun enterRoom() {  
        narrate("The villagers rally and cheer as the hero enters")  
        ringBell()  
    }  
  
    fun ringBell() {  
        narrate("The bell tower announces the hero's presence: $bellSound")  
    }  
}
```

TownSquare has access to all the properties and functions defined both in **TownSquare** and in **Room**. But **Room** does not have access to the properties and functions declared in **TownSquare**, like **ringBell**.

Test the **enterRoom** function by updating the `currentRoom` variable in `NyetHack.kt` to create an instance of **TownSquare**.

Listing 15.10 Calling subclass function implementation (NyetHack.kt)

```
fun main() {  
    narrate("Welcome to NyetHack!")  
    val playerName = promptHeroName()  
    player = Player(playerName)  
    // changeNarratorMood()  
}
```



```

player.prophesize()

var currentRoom: Room = Room("The Foyer") TownSquare()
val mortality = if (player.isImmortal) "an immortal" else "a mortal"
narrate("${player.name} of ${player.hometown}, ${player.title},
        is in ${currentRoom.description()}")
narrate("${player.name}, $mortality, has ${player.healthPoints} health points")
currentRoom.enterRoom()

player.castFireball()
player.prophesize()
}
...

```

Run NyetHack again. You should see the following output in the console:

```

...
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
Madrigal, a mortal, has 100 health points
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
A glass of Fireball springs into existence (x2)
...

```

Notice that the `currentRoom` variable’s type in `NyetHack.kt` is still **Room**, despite the fact that the instance itself is a **TownSquare** and its `enterRoom` function has been changed substantially from **Room**’s implementation. You explicitly declared the type of `currentRoom` to be **Room** so that it can hold any type of **Room**, even though you assigned `currentRoom` with a **TownSquare** constructor. Because **TownSquare** subclasses **Room**, this is completely valid syntax.

You can also subclass a subclass, creating a deeper hierarchy. If you created a subclass of **TownSquare** called **Piazza**, then **Piazza** would also be of type **TownSquare** and of type **Room**. The only limit to the number of levels that you can subclass is what makes sense for the organization of your codebase. (And, of course, your imagination.)

The different versions of `enterRoom`, based on the class they are called on, are an example of a concept in object-oriented programming called [polymorphism](#).

Polymorphism is a strategy for simplifying the structure of your program. It allows you to reuse functions for common features across groups of classes (like what happens when a player enters a room) and also to customize behavior for the unique needs of a class (like the cheering crowd in **TownSquare**).

When you subclassed **Room** to define **TownSquare**, you defined a new `enterRoom` implementation that overrides **Room**’s version. Now, when `currentRoom`’s `enterRoom` function is called, **TownSquare**’s version of `enterRoom` will be used – and no changes to `NyetHack.kt` were required.

Consider this function header:

```
fun drawBlueprint(room: Room)
```

drawBlueprint accepts a **Room** as its parameter. It can also accept any subclass of **Room**, because any subclass will have at least the capabilities that **Room** does. Polymorphism allows you to write functions that care only about what a class can do, not how it is implemented.

Opening up functions to be overridden is useful – but it does come with a side effect. When you override a function in Kotlin, the overriding function in the subclass is, by default, open to being overridden (as long as the subclass is marked open).

What if you do not want this to be the case? In the **TownSquare** example, say that you wanted any subclass of **TownSquare** to be able to customize its description but not what happens when the player enters.

The `final` keyword allows you to specify that a function cannot be overridden. Make **TownSquare** open, but also make the `enterRoom` function final so that no subclass can override the fact that villagers cheer when the hero enters a town square.

Listing 15.11 Declaring a function to be final (TownSquare.kt)

```

open class TownSquare : Room("The Town Square") {
    override val status = "Bustling"
    private var bellSound = "GWONG"

    final override fun enterRoom() {
        narrate("The villagers rally and cheer as the hero enters")
        ringBell()
    }

    fun ringBell() {
        narrate("The bell tower announces the hero's presence: $bellSound")
    }
}

```

Now, any subclass of **TownSquare** could provide an overriding function for **description** but not `enterRoom`, thanks to the `final` keyword.

As you saw when you first tried to override `enterRoom`, functions are final by default unless they are inherited from an open class. Adding the `final` keyword to an inherited function will ensure that it cannot be overridden, even if the class in which it is defined is open.

You have now seen how to use inheritance to share data and behavior between classes. You have also seen how to

use `open`, `final`, and `override` to customize what can and cannot be shared. By requiring the explicit use of the `open` and `override` keywords, Kotlin requires you to opt in to inheritance. This reduces the chances of exposing classes that were not meant to be subclassed and prevents you – or others – from overriding functions that were never meant to be overridden.

Type Checking

Nyethack is not a terribly complex program. But a production codebase can include many classes and subclasses. Despite your best efforts at clear naming, you may find yourself from time to time unsure of the type of a variable at runtime. The **is** operator is a useful tool that lets you query whether an object is of a certain type.

Try this out in the Kotlin REPL. Instantiate a **Room**. (You may need to import **Room** into the REPL with `import com.bignerdranch.nyethack.Room`. If you use autocomplete as you type the variable declaration, IntelliJ will do this for you.)

Listing 15.12 Instantiating a Room (REPL)

```
var room = Room("Foyer")
```

Next, query whether `room` is an instance of the **Room** class using the **is** operator.

Listing 15.13 Checking `room is Room` (REPL)

```
room is Room
true
```

The type of the object on the lefthand side of the **is** operator is checked against the type on the righthand side. The expression returns a **Boolean**: true if the types match, false otherwise.

Try another query: Check whether `room` is an instance of the **TownSquare** class.

Listing 15.14 Checking `room is TownSquare` (REPL)

```
room is TownSquare
false
```

`room` is of type **Room**, which is a parent class to **TownSquare**. But `room` is not itself a **TownSquare**.

Try another variable – this time, a **TownSquare**.

Listing 15.15 Checking `townSquare` (REPL)

```
var townSquare = TownSquare()
townSquare is TownSquare
true
```

```
townSquare is Room
true
```

`townSquare` is of type **TownSquare** and also of type **Room**. This, remember, is the idea that makes polymorphism possible.

If you need to know the type of a variable, type checking is a straightforward way to find out. You can build branching logic using type checking and conditionals – but be sure to bear in mind how polymorphism will affect that logic.

For example, create a `when` expression in the Kotlin REPL that returns **Room** or **TownSquare** depending on the type of a variable.

Listing 15.16 Type checking as a branching condition (REPL)

```
var className: String = when(townSquare) {
    is TownSquare -> "TownSquare"
    is Room -> "Room"
    else -> throw IllegalArgumentException()
}
print(className)
TownSquare
```

The first branch in this `when` expression evaluates as true, because `townSquare` is of type **TownSquare**. The second branch is also true, because `townSquare` is also of type **Room** – but that does not matter, because the first branch was already satisfied. So `TownSquare` is printed to the console.

Now reverse the order of the branches:

Listing 15.17 Type checking with reversed conditions (REPL)

```
var className: String = when(townSquare) {
    is Room -> "Room"
    is TownSquare -> "TownSquare"
    else -> throw IllegalArgumentException()
}
print(className)
Room
```

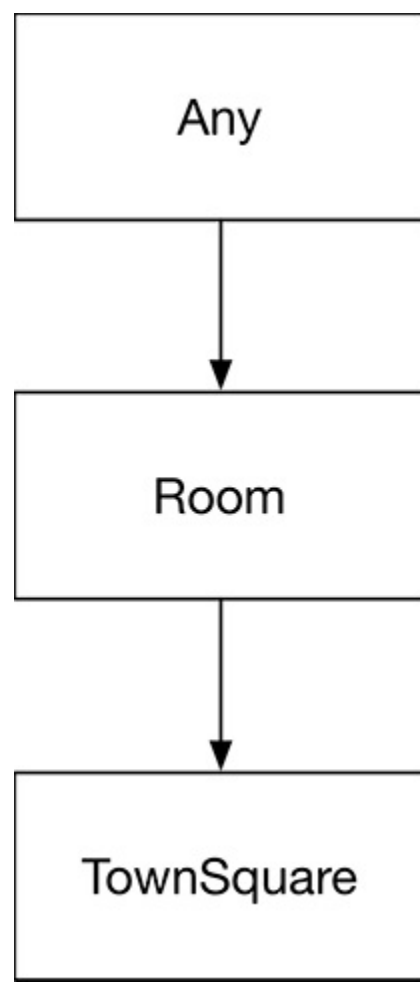
This time, `Room` is printed to the console, because the first branch evaluates to true.

When branching conditionally on object type, order matters.

The Kotlin Type Hierarchy

If you do not specify a class to extend from, your class implicitly descends from a common superclass called **Any**. This means that every type in the language eventually extends from **Any**. In the case of your **TownSquare** class, the class hierarchy looks like [Figure 15.2](#):

Figure 15.2 TownSquare type hierarchy



You can also use **Any** to define functions that accept any type of argument. Suppose that in NyetHack there are two things that can be a source of blessings: a blessed player or the room called The Fount of Blessings. A **printIsSourceOfBlessings** function, which checks and prints whether an object is a source of blessings, could take in an argument of type **Any** and use type checking to branch conditionally on the type of the argument passed to it:

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.title == "The Blessed"
    } else {
        (any as Room).name == "The Fount of Blessings"
    }
    println("$any is a source of blessings: $isSourceOfBlessings")
}
```

There are some new concepts in this code that we will discuss over the next couple of sections. For now, notice that because every object is a subclass of **Any**, you can pass arguments of whatever type you want to **printIsSourceOfBlessings**. This flexibility is useful, but it comes at the cost of not being able to immediately work with the argument. This example employs type casting, similar to what you saw in [Chapter 10](#), to get a handle on the slippery **Any** argument.

Type casting

Type casting is a way of telling Kotlin to treat an object as a different type when your program is running. This gives you the power to either lock off functionality by casting objects to a superclass's type or access functionality (like functions and properties) by casting to a subclass's type.

In the **printIsSourceOfBlessings** function, the conditional expression uses a type check to see whether **any** is of type **Player**. If it is not, then the code on the **else** branch will be executed.

The **else** branch references a **name** variable:

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.isBlessed
    } else {
        (any as Room).name == "Fount of Blessings"
    }
    println("$any is a source of blessings: $isSourceOfBlessings")
}
```

The **as** operator is used to cast values to different types. This cast says, “Treat **any** as if it were of type **Room** for purposes of this expression.” The expression in this case is a reference to **Room**’s **name** property, so that it can be compared against the string “Fount of Blessings”.

Casting is powerful and comes with great responsibility; you have to use it safely. Casting to a superclass of the original value – like **Room** to **Any** or **Int** to **Number** – is always safe. You saw another example in [Chapter 10](#) when you implicitly cast an instance of a **MutableMap** to a **Map**. As you learned in that chapter, **MutableMap** extends from **Map**.

The cast in **printIsSourceOfBlessings** works – but it is not safe. Why not? **Room**, **Player**, and **TownSquare** are the only three classes in NyetHack, so it might seem reasonable to assume that if an object is not a player it must be a type of room. But there are many classes included in the standard library and platform APIs (such as Java’s standard library and generated bindings for Kotlin/JS and Kotlin/Native users) as well as classes that you will define in the future in your own code.

Your cast will fail if the type being cast to is incompatible with the type being cast from. A **String** has nothing to do with an **Int**, for example, so a cast from **String** to **Int** would cause a **ClassCastException** that could crash your program. (Bear in mind that a cast is different from a conversion. Some strings can be converted to integers; no **String** can be cast to an **Int**.)

Casts allow you to *attempt* to cast any variable to any type, but it is up to you to make sure that the value *can* be cast to that type. You can avoid the dangers of unsafe casting with smart casting and the safe cast operator (which you will learn more about in [the section called For the More Curious: The Safe Cast Operator](#) near the end of this chapter).

If you must make an unsafe cast, be aware that your program may crash. It is best to avoid type casting unless you are sure that the cast will succeed.

Smart casting

One way to be sure that a cast will succeed is by first checking the type of the variable being cast. Return to the first branch of the conditional expression in **printIsSourceOfBlessings**.

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.isBlessed
    } else {
        (any as Room).name == "Fount of Blessings"
    }
    println("$any is a source of blessings: $isSourceOfBlessings")
}
```

The condition for entering this branch is for **any** to be of type **Player**. Inside the branch, a reference to the **isBlessed** property is made on **any**. **isBlessed** is a property defined on **Player**, not **Any**, so how is this possible without a cast?

There is, in fact, a cast happening here – a smart cast. You previously saw smart casts in action in [Chapter 7](#).

The Kotlin compiler is smart enough to recognize that if the **any is Player** type check is successful for a branch, then **any** can be treated as a **Player** within that branch. Because it knows that casting **any** to **Player** will always succeed in this branch, the compiler lets you drop the cast syntax and just reference **isBlessed**, a **Player** property, on **any**.

One way to make the unsafe casting in **printIsSourceOfBlessings** into a safe cast is to introduce another type check and smart cast, like this:

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.isBlessed
    } else if (any is Room) {
        any.name == "Fount of Blessings"
    } else {
        false
    }
    println("$any is a source of blessings: $isSourceOfBlessings")
}
```

This function will no longer crash and can now safely accept any argument regardless of its type. To accomplish this, you leveraged the intelligence of Kotlin’s compiler to automatically and safely cast your types via smart casting. Another option is to use the safe cast operator you will meet shortly.

Refactoring the Tavern

Now that you have a mechanism for expressing locations in `NyetHack`, it is a good time to revisit the Tavern. All the tavern’s behaviors are currently defined as top-level properties and functions. Spend a moment refactoring the tavern into a class that extends from **Room**.

Start by defining a new **Tavern** class inside your existing `Tavern.kt` file:

Listing 15.18 Creating a Tavern class (Tavern.kt)

```
...
private val menuItemTypes = menuData.associate { (type, name, _) ->
    name to type
}

class Tavern : Room(TAVERN_NAME) {
    override val status = "Busy"

    override fun enterRoom() {
    }
}

fun visitTavern() {
    ...
}
...
```

Next, copy and paste the implementation of **visitTavern** into **Tavern**’s **enterRoom** function:

Listing 15.19 Implementing enterRoom (Tavern.kt)

```
...
class Tavern : Room(TAVERN_NAME) {
    override val status = "Busy"

    override fun enterRoom() {
        narrate("${player.name} enters $TAVERN_NAME")
        narrate("There are several items for sale:")
        narrate(menuItems.joinToString())

        val patrons: MutableSet<String> = firstNames.shuffled()
            .zip(lastNames.shuffled()) { firstName, lastName ->
                "$firstName $lastName"
            }
            .toMutableSet()

        val patronGold: MutableMap<String, Double> = mutableMapOf(
            TAVERN_MASTER to 86.00,
            player.name to 4.50,
            *patrons.map { it to 6.00 }.toTypedArray()
        )

        narrate("${player.name} sees several patrons in the tavern:")
        narrate(patrons.joinToString())

        val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()
        narrate("The item of the day is the $itemOfDay")

        repeat(3) {
            placeOrder(patrons.random(), menuItems.random(), patronGold)
        }
        displayPatronBalances(patronGold)

        patrons
            .filter { patron -> patronGold.getOrDefault(patron, 0.0) < 4.0 }
            .also { departingPatrons ->
                patrons -= departingPatrons
                patronGold -= departingPatrons
            }
            .forEach { patron ->
                narrate("${player.name} sees $patron departing the tavern")
            }

        narrate("There are still some patrons in the tavern")
        narrate(patrons.joinToString())
    }
}
...
```

Now, copy and paste the **placeOrder** function into your new **Tavern** class.

Listing 15.20 Copying placeOrder into Tavern (Tavern.kt)

```
...
class Tavern : Room(TAVERN_NAME) {
    override val status = "Busy"

    override fun enterRoom() {
        ...
    }

    private fun placeOrder(
        patronName: String,
        menuItemName: String,
        patronGold: MutableMap<String, Double>
    ) {
        val itemPrice = menuItemPrices.getValue(menuItemName)

        narrate("$patronName speaks with $TAVERN_MASTER to place an order")
        if (itemPrice <= patronGold.getOrDefault(patronName, 0.0)) {
            val action = when (menuItemTypes[menuItemName]) {
                "shandy", "elixir" -> "pours"
                "meal" -> "serves"
                else -> "hands"
            }

            narrate("$TAVERN_MASTER $action $patronName a $menuItemName")
            narrate("$patronName pays $TAVERN_MASTER $itemPrice gold")
            patronGold[patronName] = patronGold.getValue(patronName) - itemPrice
            patronGold[TAVERN_MASTER] = patronGold.getValue(TAVERN_MASTER) + itemPrice
        } else {
            narrate("$TAVERN_MASTER says, \"You need more coin for a $menuItemName\")
        }
    }
}
...
```


With these two functions copied into the **Tavern** class, you can delete the top-level functions for **visitTavern** and **placeOrder**. (Make sure you do not delete **getFavoriteMenuItems**.)

Listing 15.21 Removing unused functions (Tavern.kt)

```
...
class Tavern : Room(TAVERN_NAME) {
    override val status = "Busy"

    override fun enterRoom() {
        ...
    }

    private fun placeOrder(
        patronName: String,
        menuItemName: String,
        patronGold: MutableMap<String, Double>
    ) {
        ...
    }
}

fun visitTavern() {
    narrate("${player.name} enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    narrate(menuItems.joinToString())
    ...
    narrate("There are still some patrons in the tavern")
    narrate(patrons.joinToString())
}
...
private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
) {
    val itemPrice = menuItemPrices.getValue(menuItemName)
    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    if (itemPrice <= patronGold.getOrDefault(patronName, 0.0)) {
        ...
    } else {
        ...
    }
}
...

```

To take better advantage of your new class, **Tavern** should store its patrons and their gold values in a class property. Make this change and remove the `patronGold` argument to **placeOrder**.

Listing 15.22 Extracting patrons and gold into properties (Tavern.kt)

```
...
class Tavern : Room(TAVERN_NAME) {
    val patrons: MutableSet<String> = firstNames.shuffled()
        .zip(lastNames.shuffled()) { firstName, lastName -> "$firstName $lastName" }
        .toMutableSet()

    val patronGold: MutableMap<String, Double> = mutableMapOf(
        TAVERN_MASTER to 86.00,
        player.name to 4.50,
        *patrons.map { it to 6.00 }.toTypedArray()
    )

    override val status = "Busy"

    override fun enterRoom() {
        narrate("${player.name} enters $TAVERN_NAME")
        narrate("There are several items for sale:")
        narrate(menuItems.joinToString())

        val patrons: MutableSet<String> = firstNames.shuffled()
        .zip(lastNames.shuffled()) { firstName, lastName ->
            "$firstName $lastName"
        }
        .toMutableSet()

        val patronGold: MutableMap<String, Double> = mutableMapOf{
            TAVERN_MASTER to 86.00,
            player.name to 4.50,
            *patrons.map { it to 6.00 }.toTypedArray()
        }

        narrate("${player.name} sees several patrons in the tavern:")
        narrate(patrons.joinToString())

        val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()
        narrate("The item of the day is the $itemOfDay")

        repeat(3) {
            placeOrder(patrons.random(), menuItems.random(), patronGold)
        }
        ...
    }

    private fun placeOrder(
        patronName: String,
        menuItemName: String,
        patronGold: MutableMap<String, Double>
    ) {
        ...
    }
}
...

```

Make sure that your refactoring is successful and has not changed the behavior of the tavern by having Madrigal enter the tavern instead of the town square when starting NyetHack.

Listing 15.23 Testing the refactored Tavern (NyetHack.kt)

```
...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    player.prophecySize()

    var currentRoom: Room = TownSquare() Tavern()
    ...
}
...

```


Run NyetHack. You should see your familiar tavern output:

```
...
Madrigal of Neversummer, The Renowned Hero, is in Taernyl's Folly
(Currently: Busy)
Madrigal, a mortal, has 100 health points
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
Madrigal sees several patrons in the tavern:
Mordoc Downstrider, Alex Baggins, Sophie Fernsworth, Tariq Ironfoot
The item of the day is Bite of Lembas Bread
Mordoc Downstrider speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Iced Boilermaker"
Sophie Fernsworth speaks with Taernyl to place an order
Taernyl pours Sophie Fernsworth a Dragon's Breath
Sophie Fernsworth pays Taernyl 5.91 gold
Alex Baggins speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Iced Boilermaker"
Madrigal intuitively knows how much money each patron has
Taernyl has 91.91 gold
Madrigal has 4.50 gold
Mordoc Downstrider has 6.00 gold
Alex Baggins has 6.00 gold
Sophie Fernsworth has 0.09 gold
Tariq Ironfoot has 6.00 gold
Madrigal sees Sophie Fernsworth departing the tavern
There are still some patrons in the tavern
Mordoc Downstrider, Alex Baggins, Tariq Ironfoot
A glass of Fireball springs into existence (x2)
Madrigal thinks about their future
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some
day form an unlikely bond between two warring factions"
```

In the next chapter, you will add functionality that lets Madrigal roam from room to room. She will be able to enter a room multiple times to her heart’s content. Some of the behaviors you implemented need to be tweaked to adjust for this reality. To begin, the item of the day should not change on a given day – for NyetHack’s in-game time, that means it should not change until you quit and restart NyetHack. Move the `itemOfDay` variable out of the **enterRoom** function to ensure its value is remembered. Also move your item of the day message so that it prints with the menu.

Listing 15.24 Remembering the item of the day (Tavern.kt)

```
...
class Tavern : Room(TAVERN_NAME) {
    ...
    val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()

    override val status = "Busy"

    override fun enterRoom() {
        narrate("${player.name} enters $TAVERN_NAME")
        narrate("There are several items for sale:")
        narrate(menuItems.joinToString())
        narrate("The item of the day is the $itemOfDay")

        narrate("${player.name} sees several patrons in the tavern:")
        narrate(patrons.joinToString())

        val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()
        narrate("The item of the day is the $itemOfDay")
        ...
    }
    ...
}
...
```

The tavern’s output is also quite a lot to take in. To streamline NyetHack’s output, have only one patron place an order when Madrigal enters the tavern. You have also proven that cash is flowing correctly through Taernyl’s Folly, so Madrigal’s omniscient accounting services are no longer required. Remove **displayPatronBalances**, as you will not need it any longer.

Madrigal should also never walk into an empty tavern, which may happen if enough patrons leave the tavern after running out of funds. To prevent this from happening, remove the logic that causes patrons to leave when they are low on gold.

Listing 15.25 Upkeep in the tavern (Tavern.kt)

```
...
class Tavern : Room(TAVERN_NAME) {
    ...
    override fun enterRoom() {
        narrate("${player.name} enters $TAVERN_NAME")
        narrate("There are several items for sale:")
        narrate(menuItems.joinToString())
        narrate("The item of the day is $itemOfDay")

        narrate("${player.name} sees several patrons in the tavern:")
        narrate(patrons.joinToString())

        repeat(3){
        placeOrder(patrons.random(), menuItems.random())
        }
        displayPatronBalances()

        patrons
            .filter { patron -> patronGold.getOrDefault(patron, 0.0) <= 4.0 }
            .also { departingPatrons ->
                patrons = departingPatrons
                patronGold = departingPatrons
            }
            .forEach { patron ->
                narrate("${player.name} sees $patron departing the tavern")
            }

        narrate("There are still some patrons in the tavern")
        narrate(patrons.joinToString())
        ...
    }
    ...
}
...
private fun displayPatronBalances(patronGold: Map<String, Double>){
    narrate("${player.name} intuitively knows how much money each patron has")
    patronGold.forEach { (patron, balance) ->
        narrate("$patron has ${"%2f".format(balance)} gold")
    }
}
}
```

Run NyetHack after making these changes. Your output will now look like this:

```
...
Madrigal, a mortal, has 100 health points
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump, Iced
  Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
The item of the day is Iced Boilermaker
Madrigal sees several patrons in the tavern:
Mordoc Downstrider, Tariq Ironfoot, Alex Baggins, Sophie Fernsworth
Tariq Ironfoot speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Iced Boilermaker"
A glass of Fireball springs into existence (x2)
Madrigal thinks about their future
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some
  day bring the gift of creation back to the world"
```

And with that, your tavern has been refactored to use the new **Room** class. Wondering why you kept **getFavoriteMenuItems** and most of the tavern’s properties at the file level instead of moving them into the class? They are outside the class because they do not change with the implementation of the tavern. Every instance of Taernyl’s Folly should have the same menu items, and patrons’ favorites stay the same regardless of which particular tavern a patron visits.

Arguably, you could move the menu properties and **getFavoriteMenuItems** function into the **Tavern** class. This is completely valid, and would probably make sense if you were writing the **Tavern** class from the beginning (instead of going through this refactoring step). Either way is perfectly valid, and you are free to choose whichever organization you prefer. You can freely mix top-level declarations and classes in the same file, which gives you many options for grouping your code.

In this chapter, you have seen how to use subclassing to share behavior between classes. In the next chapter, you will work with more types of classes, including data classes, enums, and object – Kotlin’s single-instance class – as you add a game loop to NyetHack.

For the More Curious: Any

When you print the value of a variable to the console, a function called **toString** is called to determine what that value looks like in the console. For some types, this is easy – for example, the value of a string makes sense to represent a **String** value. For other types, this is a bit less clear.

Any provides abstract definitions for common functions like **toString**, which are backed by an implementation found on the platform that your project targets.

A peek at the source for the **Any** class yields the following:

```
/**
 * The root of the Kotlin class hierarchy.
 * Every Kotlin class has [Any] as a superclass.
 */
public open class Any {
    public open operator fun equals(other: Any?): Boolean
    public open fun hashCode(): Int
    public open fun toString(): String
}
```

No definition of the **toString** function is contained in the class definition. So where is it defined, and what is returned when the **toString** function for, say, a **Player** is called?

Recall that the last line of **printIsSourceOfBlessings** prints to the console:

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.isBlessed
    } else {
        (any as Room).name == "Fount of Blessings"
    }

    println("$any is a source of blessings: $isSourceOfBlessings")
}
```

The result of calling **printIsSourceOfBlessings** and passing it a blessed player looks something like this:

```
Player@71efa55d is a source of blessings: true
```

Player@71efa55d is the result of the default implementation of **toString** on the **Any** class. Kotlin uses this implementation on the JVM and for Native targets (Kotlin/JS would show [object object] instead). You can override **toString** in your **Player** class to return something more human-readable.

The **Any** type is one of the ways that Kotlin allows for platform independence – it provides an abstraction above the class that represents a common superclass on each specific platform, like the JVM. So while **Any**'s **toString** implementation is **java.lang.Object.toString** when targeting the JVM, it could be something entirely different when compiling down to JavaScript.

This abstraction means that you do not need to know the details of each platform that your code could be run on. Instead, you can simply rely on the **Any** type.

For the More Curious: The Safe Cast Operator

Earlier in this chapter, we introduced the **as** operator as a mechanism for performing casting. We also mentioned that type casting in this way is sometimes deemed unsafe and can lead to **ClassCastException**s if you perform an invalid cast when your program runs.

In addition to the **as** operator, there is also a [safe cast operator](#), which is **as?**. **as?** behaves very similarly to **as**, but with one key difference: If you perform an invalid cast, **as?** returns null instead of throwing an exception. Try it out for yourself in the REPL:

Listing 15.26 Safe and unsafe casting (REPL)

```
5 as String
ClassCastException: class Integer cannot be cast to class String

5 as? String
null
```

as? is a great way of combining a type check and cast in one statement. If the cast fails, you can use your familiar null safety techniques from [Chapter 7](#) to provide fallback behavior instead of crashing.

Objects, Data Classes, and Enums

In the last three chapters, you learned how to use object-oriented programming principles to build meaningful relationships between objects. Despite the variety of ways they can be initialized, all the classes that you have worked with thus far have been declared with the same `class` keyword. This chapter introduces *object declarations* as well as other types of classes: *nested classes*, *data classes*, and *enum classes*. As you will see, each has its own declaration syntax and unique characteristics.

By the end of this chapter, your hero will be able to walk from room to room around the world of NyetHack – and your program will be well organized to support the enhancements coming in later chapters.

The object Keyword

In [Chapter 14](#), you learned about constructing classes. A class constructor returns an instance of a class, and you can call the constructor any number of times to create any number of instances.

For example, NyetHack can have any number of players, because **Player**'s constructor can be called as many times as you would like. For **Player**, this is desirable, because the world of NyetHack is big enough for multiple players.

But suppose you wanted a **Game** class to keep track of the state of the game. Having multiple instances of **Game** would be a problem, because they could each hold their own states, which could potentially get out of sync.

If you need to hold on to a single instance with state that is consistent throughout the time that your program is running, consider defining a [singleton](#). With the object keyword, you specify that a class will be limited to a single instance – a singleton. The first time you access an object, it is instantiated for you. That same instance will persist as long as your program is running, and each subsequent access will return the original instance.

There are three ways to use the object keyword: [object declarations](#), [object expressions](#), and [companion objects](#). We will outline the uses for each in the next three sections.

Object declarations

Object declarations are useful for organization and state management, especially when you need to maintain some state consistently throughout the lifespan of your program. You are going to define a **Game** object to do just that.

Defining a **Game** class using an object declaration will also give you a convenient place to define a game loop and will allow you to clean up the **main** function in `NyetHack.kt`. And breaking code out into classes and object declarations furthers your quest for a codebase that remains organized at scale.

Define a **Game** object in `NyetHack.kt` using an object declaration.

Listing 16.1 Declaring the Game object (NyetHack.kt)

```
...
fun main() {
    ...
}

private fun promptHeroName(): String {
    ...
}

object Game {
}
```

For this example, we have opted to add **Game** to the existing `NyetHack.kt` file, since it is important scaffolding that affects how your game runs. But if you wanted to, you could have placed **Game** in its own file. (If you did this, we would recommend calling the file `Game.kt`.) Because of Kotlin's flexibility in what can appear inside a file, this choice is yours.

The **main** function in `NyetHack.kt` should now serve exclusively to kick off gameplay. All game logic will be encapsulated in the **Game** object, of which there will be only one instance.

Because an object declaration is instantiated for you, you do not add a custom constructor with code to be called at initialization. Instead, you need an initializer block for any code that you want to be called when your object is initialized. Add one to the **Game** object with a greeting to be printed to the console when the object is instantiated.

Listing 16.2 Adding an init block to Game (NyetHack.kt)

```
object Game {
    init {
        narrate("Welcome, adventurer")
    }
}
```

Run NyetHack. Your welcome message does not print, because **Game** has not been initialized. And **Game** has not been initialized because it has not been referenced yet.

An object declaration is referenced by one of its properties or functions. To trigger **Game**'s initialization, you will define and call a function called **play**. **play** will serve as the home of the game loop for NyetHack.

Add **play** to **Game** and call it from **main**. When you call a function defined in an object declaration, you call it using the name of the object in which it is defined – not on an instance of a class, as you do for other class functions.

Listing 16.3 Calling a function defined on an object declaration (NyetHack.kt)

```
...
fun main() {
    ...
    player.castFireball()
    player.prophesize()

    Game.play()
}

private fun promptHeroName(): String {
    ...
}

object Game {
    init {
        narrate("Welcome, adventurer")
    }

    fun play() {
        while (true) {
            // Play NyetHack
        }
    }
}
```

The **Game** object will do more than encapsulate the game state; it will also hold the game loop in order to take commands from the player. Your game loop takes the form of a `while` loop that will make `NyetHack` more interactive. The `while` loop has a simple condition: `true`. This will keep the game loop running as long as your application is running.

For now, **play** does not do anything. Eventually, it will define `NyetHack`'s gameplay in terms of rounds: For each round, the player's status and other information describing the world will be printed to the console. Then, user input will be accepted via the **readLine** function.

Take a look at the game logic that is in **main** and think about where it should go in **Game**. For example, you will not want to create a new `currentRoom` at the beginning of each round, so this aspect of game logic belongs in **Game**, but not in **play**. Declare `currentRoom` as a private property of **Game**. Then move the narration and **enterRoom** calls into the **play** function's `while` loop, because you want this logic to run as part of the game to alert the player of what is happening in their surroundings.

Listing 16.4 Moving room logic into Game (NyetHack.kt)

```
...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    player.prophesize()

    var currentRoom: Room = Tavern()
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name} of ${player.hometown}, ${player.title},
    is in ${currentRoom.description()}")
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")
    currentRoom.enterRoom()

    player.castFireball()
    player.prophesize()

    Game.play()
}

private fun promptHeroName(): String {
    ...
}

object Game {
    private var currentRoom: Room = TownSquare()

    init {
        narrate("Welcome, adventurer")
    }

    fun play() {
        while (true) {
            // Play NyetHack
            narrate("${player.name} of ${player.hometown}, ${player.title},
                is in ${currentRoom.description()}")
            currentRoom.enterRoom()
        }
    }
}
```

Moving code from **main** to the **play** function in **Game** keeps the code that is essential for setting up the game loop encapsulated within the **Game** object.

main still does a couple things: It initializes the player, changes the narrator's mood, prophesizes twice, describes the health of the player, and casts fireball.

The initialization of the player and the narrator's mood can stay in **main**, but the player should not be casting fireballs or thinking about their future outside the game loop. Similarly, the narration about the player's health should happen when the game is starting – inside **Game**'s `init` block. Remove the calls to **castFireball** and **prophesize**, then move the player's health narration into **Game** to finish isolating your gameplay logic in the new class.

Listing 16.5 Tidying up main (NyetHack.kt)

```
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    player.prophesize()
```



```

        val mortality = if (player.isImmortal) "an immortal" else "a mortal"
        narrate("${player.name}, $mortality, has ${player.healthPoints} health points")

        player.castFireball()
        player.prophesize()

        Game.play()
    }
    ...
    object Game {
        ...
        init {
            narrate("Welcome, adventurer")
            val mortality = if (player.isImmortal) "an immortal" else "a mortal"
            narrate("${player.name}, $mortality, has ${player.healthPoints} health points")
        }
        ...
    }
}

```

If you were to run `NyetHack.kt` right now, it would loop indefinitely, as there is nothing to stop the loop. The last step for the game loop, at least for now, is to accept user input from the console using the **readLine** function. Recall that **readLine** pauses execution while it waits for user input in the console, then resumes execution, returning the input that was collected.

Add a call to **readLine** in your game loop to accept user input.

Listing 16.6 Accepting user input (NyetHack.kt)

```

...
object Game {
    ...
    fun play() {
        while (true) {
            narrate("${player.name} of ${player.hometown}, ${player.title},
                is in ${currentRoom.description()}")
            currentRoom.enterRoom()

            print("> Enter your command: ")
            println("Last command: ${readLine()}")

        }
    }
}

```

Try running `NyetHack` now and entering a command:

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: fight
Last command: fight
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:

```

The entered text is displayed back to you. This is excellent – new input is being sent to the game.

Object expressions

Defining a class using the `class` keyword is useful in that it establishes a new concept in your codebase. By writing a class called **Room**, you communicate that rooms exist in `NyetHack`. And by writing a subclass of **Room** called **TownSquare**, you establish that there can be specific types of rooms called town squares.

But defining a new class that can be used anywhere in your project is not always necessary. Perhaps you need a class instance that will be used for a one-off purpose. In fact, it may be so temporary that it does not even require a name.

This is another use for the object keyword: an object expression. Look at this example:

```

val abandonedTownSquare = object : TownSquare() {
    override fun enterRoom() {
        narrate("The hero anticipated applause, but no one is here...")
    }
}

```

This object expression defines a subclass of **TownSquare** – much like how you defined subclasses in [Chapter 13](#) – and returns an instance of it. Your new subclass overrides the **enterRoom** function so that no one cheers the hero’s entrance. The body of this object expression works the same as a class body, letting you override or create new functions and properties as desired.

This class follows many of the same rules as an object class. Although object expressions are not singletons, they are still one-off classes. You cannot instantiate a second instance of this object, because you do not have access to its constructor – or even a corresponding class.

Object expressions create a type of class known as an *anonymous class*, a very similar concept to anonymous functions (which we have been calling lambdas). Because anonymous classes are defined without the `class` keyword, you cannot use them as a type. You also cannot access properties or functions declared on an anonymous class outside the function where it was created.

An object declaration, on the other hand, defines a new type for the corresponding singleton. Remember your **Game**

object: You can access its functions and properties anywhere in your code (unless you are limited by a visibility modifier).

Object expressions also have a much smaller scope than an object class, and as a result an object expression takes on some of the attributes of where it is declared. If declared at the file level, an object expression is initialized immediately. If declared within another class, it is initialized when its enclosing class is initialized.

Companion objects

If you would like to add behavior to a class that can be accessed with or without an instance of the class, look no further than a companion object. Companion objects are declared within another class declaration using the companion modifier. A class can have no more than one companion object.

A companion object defines a singleton, much like the object class you saw before. When a class has a companion object, the class can behave both like a regular class and like an object class.

Take a look at the following example of a companion object defined for the **Player** class:

```
class Player(...) {
    constructor(saveFileBytes: ByteArray) : this(...)

    companion object {
        private const val SAVE_FILE_NAME = "player.dat"

        fun fromSaveFile() = Player(File(SAVE_FILE_NAME).readBytes())
    }
}
```

This **Player** class has a companion object with one function called **fromSaveFile**. If you were to call **fromSaveFile** from elsewhere in your codebase, you would do so without needing an instance of **Player**, like so:

```
val player = Player.fromSaveFile()
```

Companion objects can also have initialization logic of their own, if necessary. A companion object’s initializer will be called when its enclosing class is initialized or if one of its functions or properties is accessed directly. But no matter how many times **Player** is instantiated, there will only ever be one instance of its companion object.

[Table 16.1](#) compares the three ways objects can be defined in your code using the object keyword.

Table 16.1 Uses of the object keyword

Syntax	Description
<pre>object Game { val player: Player = ... } Game.player</pre>	Object declarations can appear wherever you can declare classes. Object declarations define a singleton class. This is useful when you want to encapsulate a behavior into a class but only want one instance of the class to ever exist.
<pre>val singleUseRoom = object : Room(name = "Pocket Dimension") { override fun enterRoom() { narrate("Madrigal doesn't think she's in Kronstadt anymore") } } singleUseRoom.enterRoom() <i>Madrigal doesn't think she's in Kronstadt anymore</i></pre>	Object expressions are typically used as function arguments or assignments to variables. Object expressions define and instantiate a single-use class that extends another type. This is useful if you want to create an instance of a class and override some of its behavior without extracting a full class.
<pre>class SpellBook(val spells: List<String> { companion object { fun createDefault(): SpellBook = SpellBook(listOf("Thundersurge", "Arcane Ammunition", "Reverse Damage")) } }) val spells = SpellBook.createDefault()</pre>	Companion objects are defined inside another class. Companion objects are singleton classes that are associated with another class. If you want a class that you can obtain multiple instances of but also need to define behaviors that can be accessed without an instance of the class, you can define those global behaviors in a companion object.

Understanding the differences in how and when object declarations, object expressions, and companion objects are instantiated is key to understanding when to use them. And using them effectively helps you write well-organized code that scales well.

Nested Classes

Not all classes defined within other classes are declared as objects. You can also use the `class` keyword to define a regular class *nested* inside another class. In this section, you will define a new **GameInput** class nested within the **Game** object.

Now that you have defined a game loop, you will want to apply some control over the user input passed to your game. NyetHack is a text adventure, driven by the user entering commands to the **readLine** function. There are two things you need to ensure about the user’s commands: First, that they are valid commands. Second, that commands with multiple parts, like “move east,” are handled correctly: You want “move” to trigger a **move** function and “east” to provide the **move** function a direction to move in.

You will address these two requirements next, starting with separating multipart commands. The **GameInput** class will provide a place for the logic that delineates command and argument.

Create a private class within the **Game** object to provide this abstraction:

Listing 16.7 Defining a nested class (NyetHack.kt)

```
...
object Game {
    ...
    private class GameInput(arg: String?) {
        private val input = arg ?: ""
        val command = input.split(" ")[0]
        val argument = input.split(" ").getOrNull(1) { "" }
    }
}
```

Why nest **GameInput** privately within **Game**? The **GameInput** class is only relevant to **Game**; it does not need to be accessed from anywhere else in NyetHack. Making **GameInput** a private, nested class means that **GameInput** can be used within **Game** but does not clutter the rest of your API.

You define two properties on the **GameInput** class: one for the command and the other for the argument. To do this, you call **split** to break the input at the space character, then **getOrNull** to attempt to fetch the second item in **split**’s resulting list. If the index you provide to **getOrNull** does not exist, **getOrNull** returns an empty string as a default.

Now you can separate multipart commands. It is time to start building the infrastructure that will handle the commands.

To act on user input, you will use a `when` expression to build a collection of valid commands in **Game**. Add a function to **GameInput** called **processCommand**. This function will use a `when` expression to branch off the command entered by the user.

You will come back to handling commands later; for now, start this function by adding a fallback case for when the user enters invalid input. Be sure to sanitize the user’s input by calling **lowercase** on the entered command.

Listing 16.8 Defining a function in a nested class (NyetHack.kt)

```
...
object Game {
    ...
    private class GameInput(arg: String?) {
        private val input = arg ?: ""
        val command = input.split(" ")[0]
        val argument = input.split(" ").getOrNull(1) { "" }

        fun processCommand() = when (command.lowercase()) {
            else -> narrate("I'm not sure what you're trying to do")
        }
    }
}
```

Now it is time to put **GameInput** to work. Replace your **readLine** call in **Game.play** with a version that uses your **GameInput** class.

Listing 16.9 Using GameInput (NyetHack.kt)

```
...
object Game {
    ...
    fun play() {
        while (true) {
            narrate("${player.name} of ${player.hometown}, ${player.title},
                is in ${currentRoom.description()}")
            currentRoom.enterRoom()

            print("> Enter your command: ")
            println("Last command: ${readLine()}")
            GameInput(readLine()).processCommand()
        }
    }
    ...
}
```

Run NyetHack. (If you normally use the run button in the toolbar at the top of the IntelliJ screen, you may notice that it is not there. This is because the game is still waiting for input from the last time you ran it. You can use the

stop and rerun button  that has replaced the run button in the toolbar or the run button next to **main**. In the pop-up that appears, choose Stop and rerun.)

Now, any input that you enter triggers the unknown command response:

```
Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
  (Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: fight
I'm not sure what you're trying to do
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
  (Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:
```

This is progress: You have restricted input to only the commands specified in a small (empty, for now) set of known inputs. Later in this chapter, you will add the “move” command, and **GameInput** will become a bit more useful.

But your hero needs a world that consists of more than one town square before they can move around the world of NyetHack.

Data Classes

Step one in building a world for your hero is to establish a coordinate system to move around on. The coordinate system will use cardinal directions to communicate direction as well as a class to represent change in direction, called **Coordinate**.

Coordinate is a simple type and a good candidate to be defined as a [data class](#). As the name suggests, data classes are classes designed specifically for holding data, and they come with some powerful data manipulation benefits, as you will see shortly.

Create a new file called `Navigation.kt` and add **Coordinate** as a data class, using the `data` keyword. **Coordinate** will have two properties defined in the constructor: an `x` coordinate and a `y` coordinate.

Listing 16.10 Defining a data class (`Navigation.kt`)

```
data class Coordinate(val x: Int, val y: Int)
```

To keep track of where the player is on the world map, add a property called `currentPosition` to the **Game** object.

Listing 16.11 Tracking player position (`NyetHack.kt`)

```
...
object Game {
    private var currentRoom: Room = TownSquare()
    private var currentPosition = Coordinate(0, 0)
    ...
}
```

Recall from [Chapter 15](#) that all classes in Kotlin ultimately inherit from **Any**. Defined on **Any** are a series of functions that you can call on any instance. These functions include **toString**, **equals**, and **hashCode**.

Any provides default implementations for all these functions, but – as you have seen before – the default behaviors are often not very helpful. Data classes provide implementations for these functions that may work better for your project. In this section, we will walk through these functions and some of the other benefits of using data classes to represent data in your codebase.

toString

The default **toString** implementation for a class is not very human readable. Take **Player**, for example.

Player is defined as a normal class, and calling **toString** on an instance on Kotlin/JVM and Kotlin/Native will return something like this:

```
Player@3527c201
```

The default implementation returns a string in the form of `ClassName@hashCode` – the name of the class followed by an effectively random hex number.

You can override **toString** in your class to provide an implementation, just like any other open function. But data classes save you that work by providing their own default implementation. For **Coordinate**, that implementation creates strings that look like this:

```
Coordinate(x=1, y=0)
```

Because `x` and `y` are properties declared in **Coordinate**'s primary constructor, they are used to represent **Coordinate** in textual form. (Properties declared outside the constructor are not included in this output.) A data class's **toString** implementation is considerably more useful than the default implementation on **Any**.

equals and hashCode

What do you think the result of the following expression would be?

```
Room("The Haunted Mines") == Room("The Haunted Mines")
```

You may be surprised, but the answer is `false`. By default, instances of a class are compared by their references, because that is the default implementation of the **equals** function in **Any**. Because the two values in this example are separate instances, they will have different references and will not be equal.

Perhaps you want to consider two rooms to be equal if they have the same name. You can provide your own equality check by overriding **equals** in your class to determine equality based on a comparison of properties, not memory references. You have seen that classes like **String** do this to compare equality based on value.

Again, data classes take care of this for you by providing an implementation of **equals** that bases equality on the

properties declared in the primary constructor. With **Coordinate** defined as a data class, `Coordinate(1, 0) == Coordinate(1, 0)` yields a result of true, because the values of the two instances' x properties are equal, as are the values of their y properties.

Data classes also provide an implementation for the **hashCode** function. Whenever you override the **equals** function, you must also provide a corresponding override of **hashCode** to avoid introducing subtle bugs in your program. **hashCode** returns a numeric representation of the object; among other things, it is used by **Set** and **Map** for fast lookups.

There are two rules that a hash code must follow: If two objects are equal based on the **equals** function, they must have the same hash code. And an object's hash code must not change unless one of its properties is changed.

In general, we do not recommend writing your own **hashCode** implementation. When you override **equals** and need to provide a corresponding **hashCode** implementation, IntelliJ can generate an implementation for you. See [the section called *For the More Curious: Defining Structural Comparison*](#) later in this chapter for information on how to do this.

Finally, as a general rule of thumb, if you find yourself wanting an **equals** implementation, consider whether a data class would be a better fit for your class before declaring it yourself.

copy

In addition to giving you more usable default implementations of functions on **Any**, data classes also provide a function that makes it easy to create copies of objects, optionally modifying values along the way.

Coordinate is defined as a data class with read-only properties for its x and y values. It is not possible to directly modify a coordinate, so if you want to change it you must obtain a new object. You could accomplish this by calling the constructor again with the desired values, but **copy** makes this task more concise and lets you omit values you do not want to change. Thus, you can create a copy of the coordinate:

```
Create a copy at the same location:    val duplicatedCoordinate = coordinate.copy()
Create a copy at the far left of the map: val leftCoordinate = coordinate.copy(x = 0)
Create a copy at the top of the map:    val topCoordinate = coordinate.copy(y = 0)
```

Many Kotlin developers consider it a best practice to only use `val` properties inside a data class. This can prevent you from mistakes like race conditions and unexpected behaviors related to mutability. For data classes structured this way, **copy** becomes almost a necessity for modifying your application's data.

Destructuring declarations

Another benefit of data classes is that they automatically enable your class's data to be destructured.

The examples of destructuring you have seen up to this point have involved destructuring types like **Pair** and **List**, as you saw in [Chapter 11](#). Under the hood, destructuring declarations depend on the declaration of functions with names like **component1**, **component2**, etc., each declared for some piece of data that you would like to return. Data classes automatically define these functions on your behalf for each property defined in their primary constructor.

There is nothing magic about a class supporting destructuring; a data class simply does the extra work required to make the class “destructurable” for you. You can make any class support destructuring by adding component operator functions to it, like so:

```
class PlayerScore(val experience: Int, val level: Int) {
    operator fun component1() = experience
    operator fun component2() = level
}

val (experience, level) = PlayerScore(1250, 5)
```

By declaring **Coordinate** to be a data class, you are able to retrieve the properties defined in **Coordinate**'s primary constructor like so:

```
val (x, y) = Coordinate(1, 0)
```

In this example, x has a value of 1, because **component1** returns the value of the first property declared in **Coordinate**'s primary constructor. y has a value of 0, because **component2** returns the value of the second property declared in **Coordinate**'s primary constructor.

These features all weigh in favor of using data classes to represent simple objects that hold data, like **Coordinate**. Classes that are often compared or copied or have their contents printed out are especially ripe for being made data classes.

However, there are also some limitations and requirements on data classes. Data classes:

- must have a primary constructor with at least one parameter
- require each of their primary constructor parameters to be marked either `val` or `var`
- cannot be declared with the `abstract`, `open`, `sealed`, or `inner` keywords

If your class does not require the **`toString`**, **`copy`**, **`equals`**, **`hashCode`**, or **`componentN`** functions, a data class offers no benefits. And if you require a customized **`equals`** function – one that uses only certain properties rather than all properties for the comparison, for example – a data class is not the right tool, because it includes all properties in the **`equals`** function it automatically generates.

You will learn about overriding **`equals`** and other functions shortly, in [the section called *Operator Overloading*](#).

Enumerated Classes

[Enumerated classes](#), or “enums,” are a special type of class where all possible values of the class are listed – or *enumerated* – in the class body.

In NyetHack, you will use an enum to represent the set of four possible directions a player can move in – the four cardinal directions. Add an enum called **Direction** to `Navigation.kt`.

Listing 16.12 Defining an enum (`Navigation.kt`)

```
data class Coordinate(val x: Int, val y: Int)

enum class Direction {
    North,
    East,
    South,
    West
}
```

Enums are more descriptive than other types of constants, like strings. You can reference [enumerated values](#) using the name of the enum class, a dot, and the name of the type, like so:

```
Direction.East
```

(A quick note on naming conventions: Enums are generally named in either PascalCase or all-caps SNAKE_CASE. Either one is fine, and you are free to pick whichever style you prefer. Developers switching to Kotlin from Java in particular may prefer to use SNAKE_CASE to match Java’s naming conventions. For this book, we will stick to PascalCase.)

Enums can do more than just declare specific cases. To use **Direction** to represent character movement in NyetHack, you will tie each **Direction** type to the **Coordinate** change when the player moves in that direction.

Moving in the game world should modify the player’s x and y position according to the direction moved. For example, if a player moves to the east, the x position should change by 1 and the y by 0. If the player moves to the south, the x position should change by 0 and the y by 1.

Add a primary constructor to the **Direction** enum that defines a `coordinate` property. Because you add a parameter to the constructor of the enum, you will have to call that constructor when defining each enumerated value in **Direction**, providing a **Coordinate** for each one.

Listing 16.13 Defining an enum constructor (`Navigation.kt`)

```
data class Coordinate(val x: Int, val y: Int)

enum class Direction(
    private val directionCoordinate: Coordinate
) {
    North(Coordinate(0, -1)),
    East(Coordinate(1, 0)),
    South(Coordinate(0, 1)),
    West(Coordinate(-1, 0))
}
```

Enums, like other classes, can also hold function declarations.

Add a function called **updateCoordinate** to **Direction** to change the player’s location based on their movement. (Be sure to add a semicolon to separate your enumerated value declarations from your function declaration.)

Listing 16.14 Defining a function in an enum (`Navigation.kt`)

```
data class Coordinate(val x: Int, val y: Int)

enum class Direction(
    private val directionCoordinate: Coordinate
) {
    North(Coordinate(0, -1)),
    East(Coordinate(1, 0)),
    South(Coordinate(0, 1)),
    West(Coordinate(-1, 0));

    fun updateCoordinate(coordinate: Coordinate) =
        Coordinate(
            x = coordinate.x + directionCoordinate.x,
            y = coordinate.y + directionCoordinate.y
        )
}
```

You call functions on enumerated values, not on the enum class itself, so calling **updateCoordinate** will look something like this:

```
var currentPosition = Coordinate(5, 2)
currentPosition = Direction.East.updateCoordinate(currentPosition)
```

Enums also have several out-of-the box features. Much like data classes, enum classes have implementations for **equals**, **hashCode**, and **toString**. Enums also have two properties of their own: a `name` and an `ordinal`.

The `name` property of an enum represents the name of the enumerated value in your code, and the `ordinal` is an **Int** that corresponds to its position in the enum declaration – similar to an index. So, for **Direction**, `North` has

a name of "North" and an ordinal of 0, `East` has a name of "East" and an ordinal of 1, and so on. Be careful about how you use these properties in your program. It is very easy to accidentally change these values when refactoring your enum declarations, which might lead to undesired behavior.

There are also two functions you can call on an enum class itself: **`values`** and **`valueOf`**. **`values`** returns an **`Array`** of all declared values of the enum. **`valueOf`** returns the enum whose name matches the input. (If there is no such enum, an exception is thrown.) These two functions can be used to look up specific enum values for dynamic inputs, which you will do later in this chapter.

Operator Overloading

You have seen that Kotlin’s built-in types come with a range of available operations and that some types tailor those operations based on the data they represent. Take the **equals** function and its associated == operator: You can use them to check whether two instances of a numeric type have the same value, whether two strings hold the same sequence of characters, and whether instances of a data class have the same values for properties in the primary constructor. Similarly, the **plus** function and + operator add two numeric values together, append one string to the end of another, and add the elements of one list to another.

When you create your own types, the Kotlin compiler does not automatically know how to apply the built-in operators to them. What does it mean to ask whether one **Player** is equal to another, for example? When you want to use built-in operators with your custom types, you have to override the operators’ functions to tell the compiler how to implement them for your type. This is known as [operator overloading](#).

You previously reaped the benefits of operator overloading in [Chapter 9](#) and [Chapter 10](#): You did not have to directly call a function called **get** to retrieve an element from a list, because the **List** type overloads the [indexed access operator](#) [] (more commonly called the *get operator*) to let you index into the collection more easily. Kotlin’s concise syntax is built on small improvements like this (spellList[3] instead of spellList.get(3)).

Coordinate is a prime candidate for improvement via operator overloading. You move your hero through the world by adding the properties of two **Coordinate** instances together. Instead of having to define that work in **Direction**, you can overload the **plus** operator on **Coordinate**.

Make it so in Navigation.kt, prepending the function declaration with the operator modifier.

Listing 16.15 Overloading the plus operator (Navigation.kt)

```
data class Coordinate(val x: Int, val y: Int) {
    operator fun plus(other: Coordinate) = Coordinate(x + other.x, y + other.y)
}
...
```

Now, you can simply use the addition operator (+) to add two **Coordinate** instances together. Do that in **Direction**:

Listing 16.16 Using an overloaded operator (Navigation.kt)

```
data class Coordinate(val x: Int, val y: Int) {
    operator fun plus(other: Coordinate) = Coordinate(x + other.x, y + other.y)
}

enum class Direction(
    private val directionCoordinate: Coordinate
) {
    North(Coordinate(0, -1)),
    East(Coordinate(1, 0)),
    South(Coordinate(0, 1)),
    West(Coordinate(-1, 0));

    fun updateCoordinate(coordinate: Coordinate) =
        Coordinate{
            x = coordinate.x + directionCoordinate.x,
            y = coordinate.y + directionCoordinate.y
        }
        coordinate + directionCoordinate
}
```

[Table 16.2](#) shows many of the operators you can overload in Kotlin:

Table 16.2 Common operators

Operator	Function name	Purpose
+	plus	Adds an object to another.
++	inc	Increments the value of an object.
+=	plusAssign	Adds an object to another and assigns the result to the first.
-	minus	Subtracts an object from another.
--	dec	Decrements the value of an object.
-=	minusAssign	Subtracts an object from another and assigns the result to the first.
*	times	Multiplies an object by another.
/	div	Divides an object by another.
==	equals	Returns true if two objects are equal, false otherwise.
>	compareTo	Returns true if the object on the lefthand side is greater than the object on the righthand side, false otherwise.
[]	get	Returns the element in a collection at a given index.
..		

	rangeTo	Creates a range object.
in	contains	Returns true if an object exists within a collection.
()	invoke	Executes a function as if the value is a lambda expression.

These operators can be overloaded on any class, but make sure to do so only when it makes sense. While you *could* assign logic to the addition operator on the **Player** class, what would “Player plus Player” mean? Ask yourself this question before overloading an operator.

Exploring the World of NyetHack

Now that you have built a game loop and established a cardinal direction system on a coordinate plane, it is time to add more rooms to explore in NyetHack.

To set up a map of the world, you need a list that will hold all the rooms. In fact, since players can move in two dimensions, you need a list containing three lists of rooms – one for each row of rooms.

The first list of rooms will hold the Town Square (where the player begins), Tavern, and Back Room, from west to east. The second list of rooms will hold the long corridor and the generic room. The third list of rooms will only hold the dungeon. These lists will be held in a fourth list, called `worldMap`.

[Figure 16.1](#) shows the grid of rooms.

Figure 16.1 NyetHack world map

Town Square (0, 0)	Tavern (1, 0)	Back Room (2, 0)
Long Corridor (0, 1)	Generic Room (1, 1)	
Dungeon (0, 2)		

Add a `worldMap` property to **Game** with a series of rooms for the hero to explore.

Listing 16.17 Defining a world map in NyetHack (NyetHack.kt)

```
...
object Game {
    private val worldMap = listOf(
        listOf(TownSquare(), Tavern(), Room("Back Room")),
        listOf(Room("A Long Corridor"), Room("A Generic Room")),
        listOf(Room("The Dungeon"))
    )

    private var currentRoom: Room = TownSquare() worldMap[0][0]
    private var currentPosition = Coordinate(0, 0)
    ...
}
```

With the rooms in place, it is time to add the “move” command and give the player the ability to step out into the mysterious land of NyetHack. Add a function called **move** that takes in a **Direction** argument and updates `currentRoom` and `currentPosition` if the player can move in that direction.

Listing 16.18 Defining the move function (NyetHack.kt)

```
...
object Game {
    ...
    fun play() {
        while (true) {
            narrate("${player.name} of ${player.hometown}, ${player.title},
                is in ${currentRoom.description()}")
            currentRoom.enterRoom()

            print("> Enter your command: ")
            GameInput(readLine()).processCommand()
        }
    }

    fun move(direction: Direction) {
        val newPosition = direction.updateCoordinate(currentPosition)
        val newRoom = worldMap.getOrNull(newPosition.y)?.getOrNull(newPosition.x)

        if (newRoom != null) {
            narrate("The hero moves ${direction.name}")
            currentPosition = newPosition
            currentRoom = newRoom
        } else {
            narrate("You cannot move ${direction.name}")
        }
    }
}
```

```

    }
    ...
}

```

You are using the **getOrNull** function to determine whether the coordinate matches a room on the map. If the player tries to walk off the edge of the map, this lookup will return `null` and the movement should be denied.

Also, notice that this code is using the `name` property on your enum. Typically, you do not want to print this value to the console, because your enum names are not always user-facing. Using enum names like this can also make it difficult to translate your program into other languages. But NyetHack’s **Direction** is not subject to either of these precautions, so you can use `name` for simplicity.

(In more complex projects, we would recommend adding your own `directionName` property to this enum. If your program also requires internationalization, you can implement this custom property using your platform’s translation APIs to obtain the correct string based on the user’s location.)

The **move** function should be called when the player enters the “move” command, which you will now implement using the **GameInput** class you wrote earlier in this chapter:

Listing 16.19 Implementing the move command (NyetHack.kt)

```

...
object Game {
    ...
    private class GameInput(arg: String?) {
        private val input = arg ?: ""
        val command = input.split(" ")[0]
        val argument = input.split(" ").getOrNull(1) { "" }

        fun processCommand() = when (command.lowercase()) {
            "move" -> {
                val direction = Direction.values()
                    .firstOrNull { it.name.equals(argument, ignoreCase = true) }
                if (direction != null) {
                    move(direction)
                } else {
                    narrate("I don't know what direction that is")
                }
            }
            else -> narrate("I'm not sure what you're trying to do")
        }
    }
}

```

This new code starts by converting the `argument` to a **Direction**. You use the **values** function on the enum class and find a corresponding **Direction** (ignoring its case). If no direction has a matching name, the lookup will return `null`, and the narrator will express their confusion. Otherwise, you now have an instance of **Direction** that you can call **move** with.

Try running NyetHack and moving around the world. You should see some output like this:

```

...
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: move east
The hero moves East
Madrigal of Neversummer, The Renowned Hero, is in Taernyl's Folly
(Currently: Busy)
Madrigal enters Taernyl's Folly
...

```

And that is it – you are now able to walk around the world of NyetHack. In this chapter, you learned how to use several variants of classes. Beyond the `class` keyword, you can use object declarations, data classes, and enum classes to represent data. Using the right tool for the job will make the relationships among objects in your code more straightforward.

In the next chapter, you will learn about interfaces and abstract classes – mechanisms for defining protocols that your classes must adhere to – as you add the thrill of combat to NyetHack.

For the More Curious: Defining Structural Comparison

Imagine a **Weapon** class that has `name` and `type` properties:

```
open class Weapon(val name: String, val type: String)
```

Suppose you would like two individual weapon instances to be considered structurally equal, using the structural equality operator (`==`), if the values of their names and types are structurally equal. By default, as we said earlier in this chapter, `==` checks referential equality for objects, so this expression would evaluate as false:

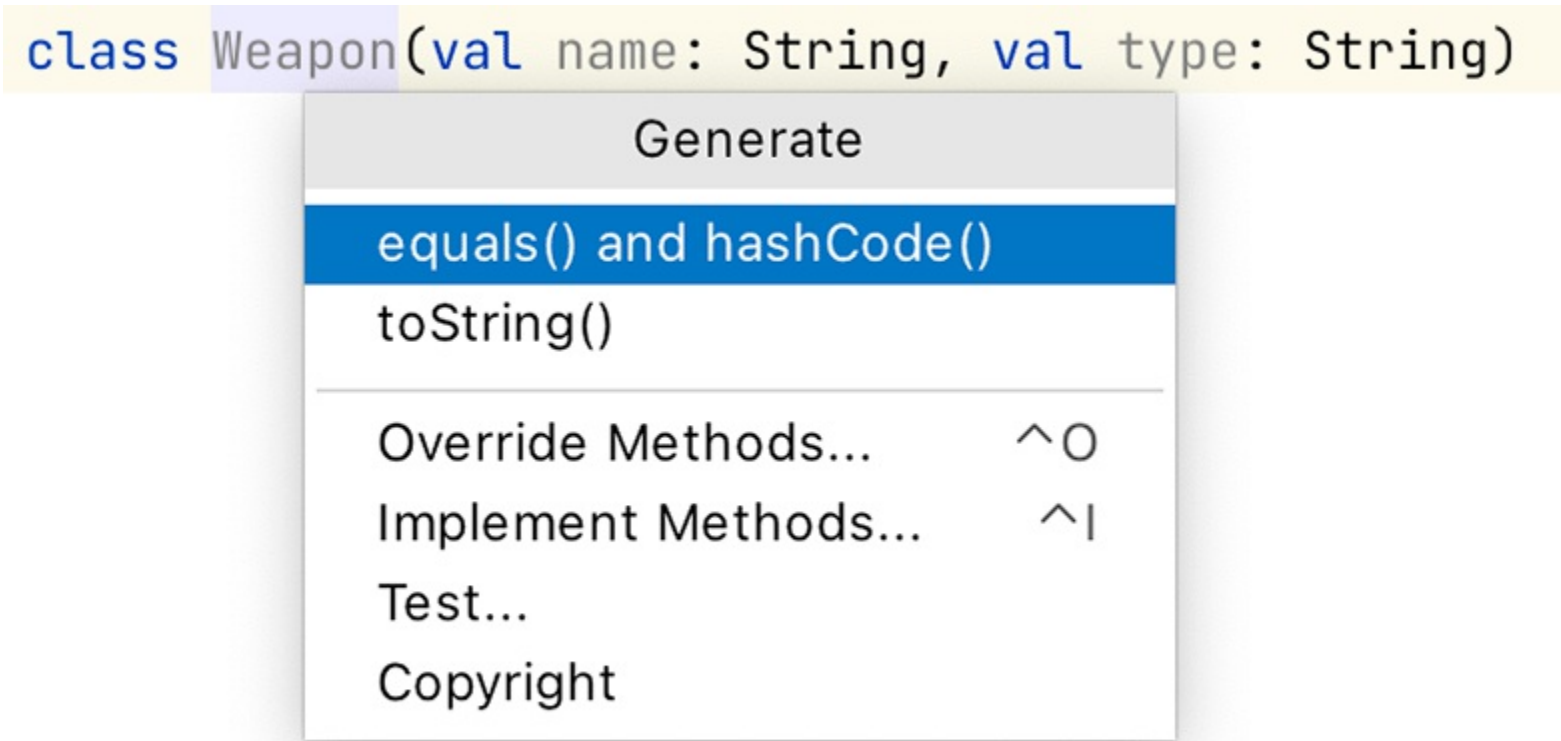
```
open class Weapon(val name: String, val type: String)
Weapon("Mjolnir", "hammer") == Weapon("Mjolnir", "hammer") // False
```

You saw in this chapter that data classes provide a solution to this problem – an implementation of **equals** that bases equality on the properties declared in the primary constructor. But **Weapon** is not (and cannot be) a data class, because it is intended to be the base class for other weapon variations (hence the `open` keyword). Data classes are not permitted to be superclasses.

However, as we discussed in [the section called Operator Overloading](#), you can provide your own implementation of **equals** and **hashCode** to specify how instances of your class should be compared structurally.

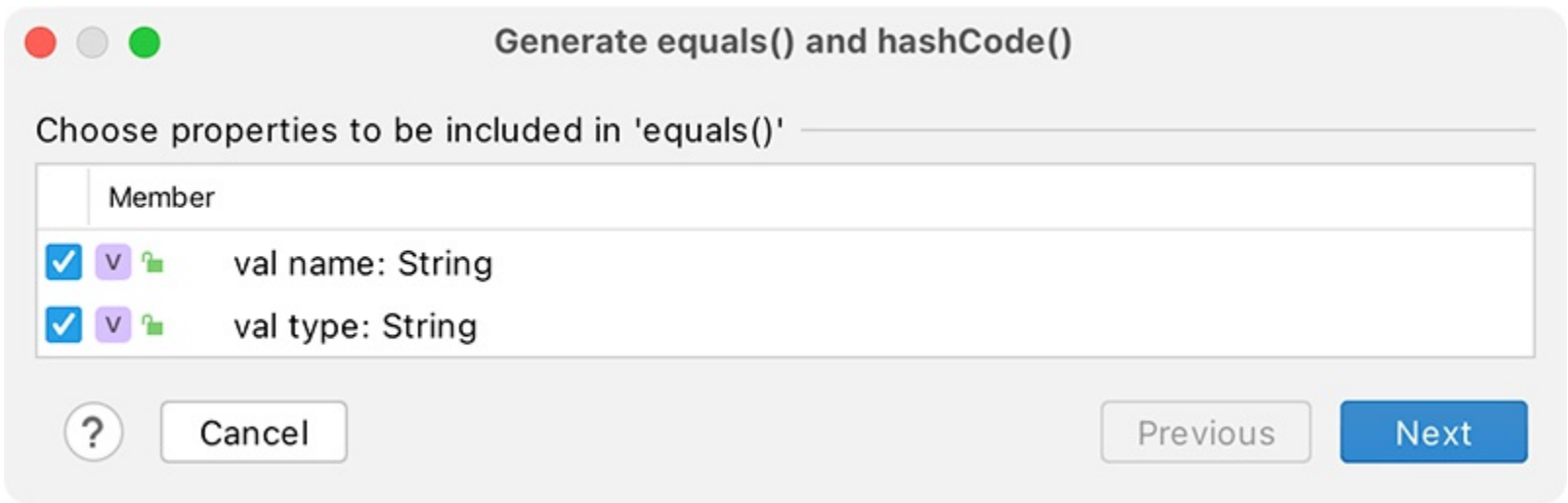
This need is so common that IntelliJ has a Generate task for adding the function overrides via its Code → Generate menu (Command-N [Alt-Insert]), which brings up the Generate dialog ([Figure 16.2](#)):

Figure 16.2 The Generate dialog



In this dialog, select `equals()` and `hashCode()`. When generating **equals** and **hashCode** overrides, you can select the properties that should be used when you compare two instances of your object structurally ([Figure 16.3](#)). Check the boxes to select both `name` and `type`.

Figure 16.3 Choosing properties to be included in equals override



IntelliJ adds the **equals** and **hashCode** functions to the class based on the choices made:

```
open class Weapon(val name: String, val type: String) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
```



```

        if (other !is Weapon) return false

        other as Weapon

        if (name != other.name) return false
        if (type != other.type) return false

        return true
    }

    override fun hashCode(): Int {
        var result = name.hashCode()
        result = 31 * result + type.hashCode()
        return result
    }
}

```

The overridden **equals** function that was generated sets up a structural comparison between the properties selected in the Generate command. If any of the properties are not structurally equal, then the comparison results in false. Otherwise, true is returned.

So, with these overrides in place, comparing two weapons would result in true as long as their names and types are the same:

```
Weapon("Mjolnir", "hammer") == Weapon("Mjolnir", "hammer") // True
```

As we mentioned earlier, whenever you override the **equals** function, you should also provide a corresponding override of **hashCode**. Generally, the properties you use in **equals** should be hashed together to form this hash code. IntelliJ does this hashing for you by taking the hash code of each property, multiplying it by an arbitrary prime number, and taking the sum.

For the More Curious: Algebraic Data Types

[Algebraic data types](#) (or ADTs, for short) allow you to represent a closed set of possible subtypes that can be associated with a given type. Enum classes are a simple form of ADT.

Imagine a **Student** class that has three possible associated states, depending on the student's enrollment status: **NotEnrolled**, **Active**, or **Graduated**.

Using the enum class that you learned about in this chapter, you could model the three states for the **Student** class as follows:

```
enum class StudentStatus {
    NotEnrolled,
    Active,
    Graduated
}

class Student(var status: StudentStatus)

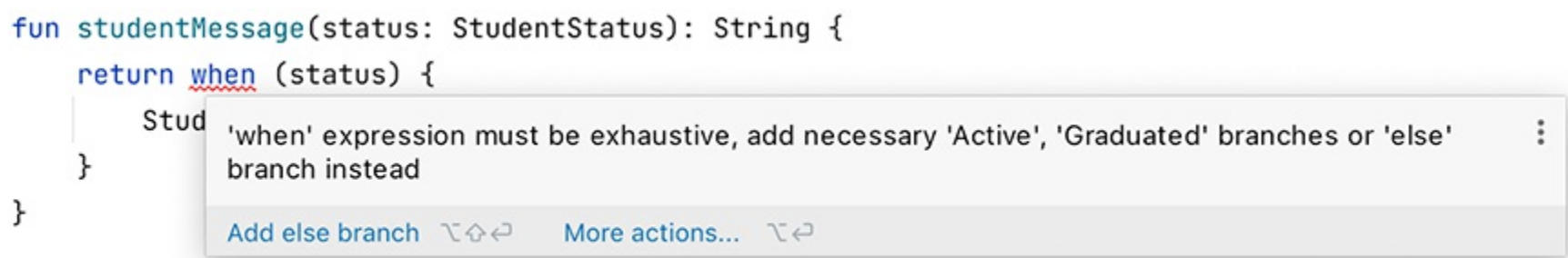
fun main() {
    val student = Student(StudentStatus.NotEnrolled)
}
```

And you could write a function that generates a student message using the student's status:

```
fun studentMessage(status: StudentStatus): String {
    return when (status) {
        StudentStatus.NotEnrolled -> "Please choose a course."
    }
}
```

One of the benefits of enums and other ADTs is that the compiler can check to ensure that you handled all possibilities, because an ADT is a closed set of possible types. The implementation for **studentMessage** does not handle the **Active** or **Graduated** types, so the compiler would give an error ([Figure 16.4](#)):

Figure 16.4 Adding necessary branches



```
fun studentMessage(status: StudentStatus): String {
    return when (status) {
        StudentStatus.NotEnrolled -> "Please choose a course."
    }
}
```

The compiler is satisfied when all types are addressed either explicitly or through an else branch:

```
fun studentMessage(status: StudentStatus): String {
    return when (status) {
        StudentStatus.NotEnrolled -> "Please choose a course."
        StudentStatus.Active -> "Welcome, student!"
        StudentStatus.Graduated -> "Congratulations!"
    }
}
```

For more complex ADTs, you can use Kotlin's [sealed classes](#) to implement more sophisticated definitions. Sealed classes let you specify an ADT similar to an enum, but with more control over the specific subtypes than an enum provides.

For example, imagine that when a student is active, the student is also assigned a course ID. You could add a course ID property to the enum definition, which might look like this:

```
enum class StudentStatus {
    NotEnrolled,
    Active,
    Graduated;
    var courseId: String? = null // Used for Active only
}
```

There are two issues with this approach. Because this property is only used in the Active case, you have two unneeded null states for the property in other cases. Also, Kotlin only creates one instance of each enumerated value, which gets shared across your program – much like the **Game** singleton you created using an object declaration. If you set up your program this way, all your students will share a single Active instance. Unless your students are all taking the same course – or you only have one student – this representation will not work.

A better solution would be to use a sealed class to model the student statuses:

```
sealed class StudentStatus {
    object NotEnrolled : StudentStatus()
    data class Active(val courseId: String) : StudentStatus()
    object Graduated : StudentStatus()
}
```

The **StudentStatus** sealed class has a limited number of subclasses. Sealed classes are only eligible to be subclassed by other classes declared in the same package and codebase as the definition of the sealed class itself.

Because of the restrictions on how sealed classes can be extended, Kotlin knows all the possible implementations at

compile time. This allows the compiler to check a when expression's exhaustiveness without requiring an else branch – much like it can for enums.

The object keyword is used for the statuses that require no course ID because there will never be any variation on their instances. But the Active status is defined as a regular class (a data class, specifically) because it will have different instances, because the course ID will change depending on the student.

Using the new sealed class in the when would allow you to read the courseId from the **Active** class, accessible through smart casting:

```
fun main() {
    val student = Student(StudentStatus.Active("Kotlin101"))
    studentMessage(student.status)
}

fun studentMessage(status: StudentStatus): String {
    return when (status) {
        is StudentStatus.NotEnrolled -> "Please choose a course!"
        is StudentStatus.Active -> "You are enrolled in: ${status.courseId}"
        is StudentStatus.Graduated -> "Congratulations!"
    }
}
```

For the More Curious: Value Classes

In addition to the other kinds of classes you have seen in this chapter, Kotlin has [value classes](#). Value classes are useful when you want to make a new class that acts as a different interpretation of an existing type. For example, you may want to have several units of distance for measuring how far away things on NyetHack’s map are. You could use value classes to define types for miles and kilometers:

```
@JvmInline
value class Kilometers(private val kilometers: Double) {
    operator fun plus(other: Kilometers) =
        Kilometers(kilometers + other.kilometers)

    fun toMiles() = kilometers / 1.609
}

@JvmInline
value class Miles(private val miles: Double) {
    operator fun plus(other: Miles) =
        Miles(miles + other.miles)

    fun toKilometers() = miles * 1.609
}
```

Having a separate type for a particular unit can be a great convenience, because it allows you to encode units in your program’s types. In this example, it also prevents you from accidentally attempting to add miles to kilometers without converting units first. Normally, classes like this come with a drawback: There is memory overhead to instantiating an instance of any class. Value classes sidestep this problem entirely.

When you use value classes in your code, they are replaced with the type they are wrapping (in this case, **Double**). All the functions on the class (like **plus**, **toMiles**, and **toKilometers**) are then compiled into static functions that can be called without an instance of the value class.

The end result is that you get the type safety of declaring custom classes that wrap existing types without any additional overhead. However, value classes are not without their limitations:

- A value class’s primary constructor must have exactly one argument, and it must be declared as a `val` property. This is the value that will be inlined wherever the class is used.
- Value classes cannot declare any additional properties with backing fields. However, computed `var` or `val` properties are allowed.
- Value classes cannot override **equals** or **hashCode**. Kotlin will use the implementations from the value being wrapped.
- Value classes cannot be marked as open and cannot extend from another class. (But they can implement interfaces, which you will learn about in the next chapter.)

Challenge: More Commands

With the changes you made in this chapter, Madrigal no longer casts fireball or prophesizes. Fix this by adding two new commands, `cast fireball` and `prophesize`, to call the corresponding functions on **Player**.

Additionally, players will most likely want to quit NyetHack at some point, and currently NyetHack offers no way to do that. Add another command for when a user enters “quit” or “exit.” When this command is received, NyetHack should display a farewell message to the adventurer and terminate. Hint: Remember that, currently, your `while` loop executes forever – a significant part of solving this challenge is to end that loop conditionally.

Challenge: Implementing a World Map

Remember when we said NyetHack would not feature awesome ASCII graphics? Once you successfully complete this challenge, it will!

Players sometimes get lost in the expansive world of NyetHack, and fortunately you have the power to give them a magic map of the realm. Implement a “map” command that displays the player’s current position in the game world. For a player currently at the tavern, the game interaction should resemble this:

```
> Enter your command: map
0 X 0
0 0
0
0
```

The X represents the room the player is currently in.

Challenge: Ring the Bell

Add a “ring” command to NyetHack so you can ring the bell as many times as you would like from within the town square.

Hint: You will have to make the **ringBell** function public.

Interfaces and Abstract Classes

In this chapter you will see how to define and use [interfaces](#) and [abstract classes](#) in Kotlin.

An interface allows you to specify common properties and behavior that are supported by a subset of classes in your program – without being required to specify how they will be implemented. This capability – the *what* without the *how* – is useful when inheritance is not the right relationship for classes in a program. Using an interface, a group of classes can have properties or functions in common without sharing a superclass or subclassing one another.

You will also work with a type of class called an abstract class, a hybrid between the features of interfaces and classes. Abstract classes are similar to interfaces in that they can specify the *what* without the *how*, but they are different in that they can also define constructors and act as a superclass.

These new concepts will allow you to add an exciting feature to NyetHack: Now that your hero can walk around, you will add a combat system to deal with the evildoers your hero encounters.

Defining an Interface

To define how combat is performed, you will first create an interface that specifies the functions and properties used for entities in the game when performing combat. Your player will face goblins, but you will define a combat system that can be applied to any type of creature – not just goblins.

Create a new file called `Creature.kt` – in the `com.bignerdranch.nyethack` package, to avoid naming collisions – and define a `Fightable` interface, using the keyword `interface`:

Listing 17.1 Defining an interface (`Creature.kt`)

```
interface Fightable {
    val name: String
    var healthPoints: Int
    val diceCount: Int
    val diceSides: Int

    fun takeDamage(damage: Int)

    fun attack(opponent: Fightable)
}
```

Your interface declaration defines things that are common to any entity that can fight in NyetHack. `Fightable` creatures use the number of dice, the number of sides on each die, and the damage roll – the sum of the numbers rolled on the dice – to determine the amount of damage dealt to an enemy. `Fightable` creatures must also have a name, `healthPoints`, and an implementation for two functions: **takeDamage** and **attack**.

The four properties in `Fightable` have no initializers, and the functions **takeDamage** and **attack** have no function body. An interface is not concerned with providing initializers or function bodies. Remember – interfaces only specify the what, not the how.

The `Fightable` interface is also the type of the `opponent` parameter that the **attack** function accepts. An interface can be used as a type for a parameter, just as a class can be used as a parameter type.

When a function specifies a parameter type, that function cares about what the argument can do, not how the behavior is implemented. This is one of the strengths of an interface – you can create a set of requirements that is shared between classes that otherwise have nothing in common.

Implementing an Interface

To use an interface, we say that you “implement” it on a class. There are two parts to this: First, you declare that the class implements the interface. Then, you must ensure that the class provides implementations for all the properties and functions specified in the interface.

Use the `:` operator to implement the `Fightable` interface on **Player**.

Listing 17.2 Implementing an interface (`Player.kt`)

```
class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    override var healthPoints: Int,
    val isImmortal: Boolean
) : Fightable {

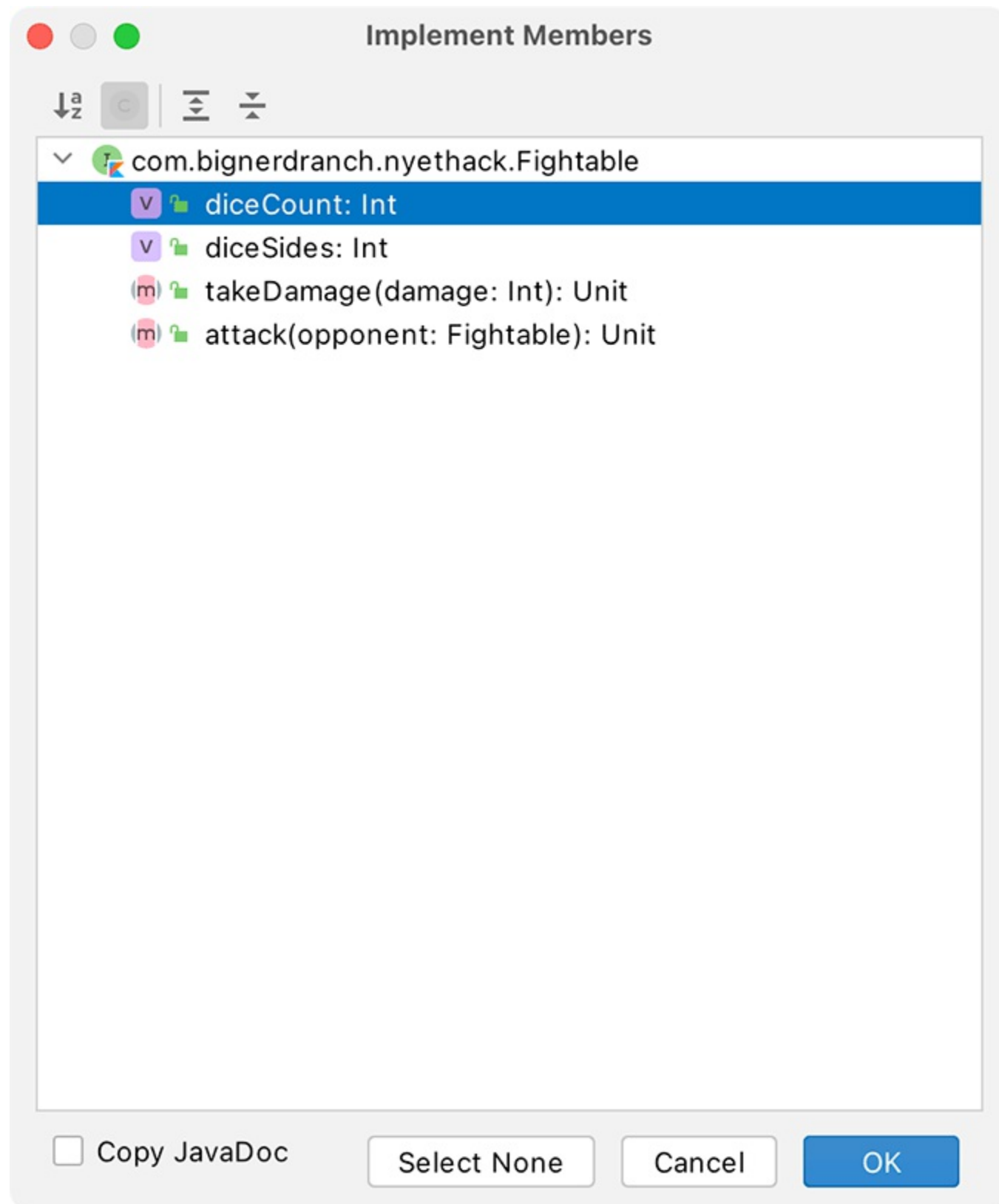
    override var name = initialName
    get() = field.replaceFirstChar { it.uppercaseChar() }
    private set(value) {
        field = value.trim()
    }
    ...
}
```

(We will explain the `override` keywords you are adding here shortly.)

When you add the `Fightable` interface to **Player**, IntelliJ indicates that functions and properties are missing. Warning you that properties and functions have yet to be implemented on **Player** helps you adhere to `Fightable`’s rules, and IntelliJ will also help you implement everything that is required by the interface.

Right-click **Player** and select `Generate... → Implement Methods...`, then select `diceCount`, `diceSides`, and **takeDamage** in the `Implement Members` dialog ([Figure 17.1](#)). (You will deal with **attack** in the next section.)

Figure 17.1 Implementing `Fightable` members



After IntelliJ generates implementations for the **Player** class, rearrange them as shown in [Listing 17.3](#) to group your properties and functions.

Listing 17.3 Generating stubs (`Player.kt`)

```
class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    override var healthPoints: Int,
    val isImmortal: Boolean
) : Fightable {
    ...
    val prophecy by lazy {
        ...
    }

    override val diceCount: Int
        get() = TODO("Not yet implemented")

    override val diceSides: Int
        get() = TODO("Not yet implemented")

    init {
        require(healthPoints > 0) { "healthPoints must be greater than zero" }
        require(initialName.isNotBlank()) { "Player must have a name" }
    }
    ...
    fun prophesize() {
        narrate("$name thinks about their future")
        narrate("A fortune teller told Madrigal, \"$prophecy\"")
    }

    override fun takeDamage(damage: Int) {
        TODO("Not yet implemented")
    }
}
```

The implementations added to **Player** are just stubs. You will flesh them out next. (By the way, you might recall the **TODO** function from the discussion of the **Nothing** type in [Chapter 4](#). Here it is in action.) Once you implement `diceCount`, `diceSides`, and **takeDamage** (and take care of **attack**), **Player** will satisfy the `Fightable` interface and can be used in combat.

The property and function implementations, including `healthPoints` and `name`, all use the `override` keyword. This might surprise you – after all, you are not replacing an implementation for these properties in `Fightable`. However, all implementations of interface properties and functions must be marked with `override`.

On the other hand, the `open` keyword is not required on function declarations in an interface. This is because all properties and functions you add to an interface must be open implicitly, since they would serve no purpose otherwise. After all, an interface outlines the what, and the how must be provided in the classes that implement it.

Replace the **TODO** calls in `diceCount`, `diceSides`, and **takeDamage** with appropriate values and functionality.

Listing 17.4 Implementing an interface (`Player.kt`)

```
class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    override var healthPoints: Int,
    val isImmortal: Boolean
) : Fightable {
    ...
    override val diceCount: Int = 3
    get() = TODO("Not yet implemented")

    override val diceSides: Int = 4
    get() = TODO("Not yet implemented")
    ...
    override fun takeDamage(damage: Int) {
        TODO("Not yet implemented")
        if (!isImmortal) {
            healthPoints -= damage
        }
    }
}
```

`diceCount` and `diceSides` are implemented with integers. **Player**'s **takeDamage** subtracts however much damage was dealt from the player's pool of health points, unless the player is immortal (in which case they cannot be harmed by ordinary means). **takeDamage** will be called by **attack** when you implement it in the next section.

Default Implementations

We have said several times now that interfaces focus on the what and not the how. You can, however, provide a default implementation for property getters and functions in an interface. Classes that implement the interface then have the option of using the default or defining their own implementation.

Provide a default implementation for **attack** in `Fightable`. This function should take the sum of the dice rolls and deal the corresponding amount of damage.

Listing 17.5 Defining a default implementation (`Creature.kt`)

```
import kotlin.random.Random

interface Fightable {
    val name: String
    var healthPoints: Int
    val diceCount: Int
    val diceSides: Int

    fun takeDamage(damage: Int)

    fun attack(opponent: Fightable) {
        val damageRoll = (0 until diceCount).sumOf {
            Random.nextInt(diceSides + 1)
        }
        narrate("$name deals $damageRoll to ${opponent.name}")
        opponent.takeDamage(damageRoll)
    }
}
```

Now that **attack** has a default implementation, any class that implements the `Fightable` interface can opt out of providing an implementation for the **attack** function. The errors have disappeared from your project; **Player** now fully implements `Fightable`. Run your program to confirm that everything works as it did before.

You can also provide default implementations for properties in an interface, but they are limited to being computed properties – an interface is not allowed to allocate space for backing properties of its own. Not every property or function needs a unique implementation in every class, so providing a default implementation is a good way to reduce duplication in your code.

Abstract Classes

Abstract classes provide another way to enforce structure in your classes. An abstract class is never directly instantiated. Its purpose is to provide function implementations through inheritance to subclasses that *are* instantiated.

An abstract class is defined by prepending the `abstract` keyword to a class definition. In addition to function implementations, abstract classes can include [abstract functions](#) – function declarations without implementations.

It is time to give the player something to fight in NyetHack. Add an abstract class called **Monster** to `Creature.kt`. **Monster** implements the `Fightable` interface, and therefore needs `name` and `healthPoints` properties and a **takeDamage** function. (What about the other `Fightable` properties? We will return to those in a moment.)

Listing 17.6 Defining an abstract class (Creature.kt)

```
interface Fightable {
    val name: String
    var healthPoints: Int
    val diceCount: Int
    val diceSides: Int

    fun takeDamage(damage: Int)

    fun attack(opponent: Fightable) {
        val damageRoll = (0 until diceCount).sumOf {
            Random.nextInt(diceSides + 1)
        }
        narrate("$name deals $damageRoll to ${opponent.name}")
        opponent.takeDamage(damageRoll)
    }
}

abstract class Monster(
    override val name: String,
    val description: String,
    override var healthPoints: Int
) : Fightable {
    override fun takeDamage(damage: Int) {
        healthPoints -= damage
    }
}
```

You define **Monster** as an abstract class because it is meant as a foundation for more specific creatures in the game. You will never create an instance of **Monster** – and could not if you tried. Instead, you will create instances of **Monster** subclasses: more specific monsters, like goblins, wraiths, or dragons, that are concrete versions of an abstract monster.

Take another look at **Monster**'s constructor. You defined the `name` and `healthPoints` properties in the constructor with the `override` keyword. These two properties are inherited from the `Fightable` interface, and they are accessible by the **Monster** class regardless of whether they appear in the class itself. You could omit these properties from your **Monster** class, but there are two advantages to including them.

The first advantage of declaring overridden properties in a constructor is that subclasses of the **Monster** class can now implement the properties by passing in a constructor value instead of declaring their own property overrides. This leads to slightly more concise syntax in your subclasses.

The second advantage is more nuanced. At some point in the future, the world may make peace with monsters, and you might want to update your **Monster** class to stop implementing the `Fightable` interface. But it is still valuable to know a monster's name and health points, even if it is a friendly monster. In this scenario, monsters would still have those properties, because they are declared by the class itself. (Just remember to delete the `override` keyword.)

Defining **Monster** as an abstract class provides a template for what it means to be a monster in NyetHack: A monster must have a name and a description, and – for now – it must satisfy the criteria of the `Fightable` interface.

Now, create the first concrete version of the **Monster** abstract class – the **Goblin** subclass – in `Creature.kt`.

Listing 17.7 Subclassing an abstract class (Creature.kt)

```
interface Fightable {
    ...
}

abstract class Monster(
    override val name: String,
    val description: String,
    override var healthPoints: Int
) : Fightable {
    override fun takeDamage(damage: Int) {
        healthPoints -= damage
    }
}

class Goblin(
    name: String = "Goblin",
    description: String = "A nasty-looking goblin",
    healthPoints: Int = 30
) : Monster(name, description, healthPoints)
```

Because **Goblin** is a subclass of **Monster**, it has all the properties and functions that **Monster** does.

If you attempted to compile your code at this point, compilation would fail. This is because both `diceCount` and `diceSides` are specified as requirements of the `Fightable` interface, but they are not implemented in **Goblin** (and have no default implementation).

But **Goblin** does not explicitly implement `Fightable`, so why does it need to implement its requirements?

A subclass shares all functionality with its superclass, by default. This is true no matter what kind of class the superclass is. If a class implements an interface, then its subclass must also satisfy the requirements of the interface.

Monster does not have to include all the requirements of the `Fightable` interface, even though it implements it, because it is an abstract class and will never be instantiated. But its subclasses must implement all requirements of `Fightable`, either through inheritance from **Monster** or on their own.

Satisfy the requirements defined on the `Fightable` interface by adding them to **Goblin**:

Listing 17.8 Implementing properties in the subclass of an abstract class (`Creature.kt`)

```
interface Fightable {
    ...
}

abstract class Monster(
    override val name: String,
    val description: String,
    override var healthPoints: Int
) : Fightable {
    ...
}

class Goblin(
    name: String = "Goblin",
    description: String = "A nasty-looking goblin",
    healthPoints: Int = 30
) : Monster(name, description, healthPoints) {
    override val diceCount = 2
    override val diceSides = 8
}
```

Run your program again to make sure it compiles as expected.

You may have noticed the similarity between abstract classes and interfaces: Both can define functions and properties that do not require an implementation. What, then, is the difference between the two?

For one thing, an interface cannot specify a constructor. Interfaces also cannot prevent inheritors from overriding default behaviors, and they have limitations on which properties and functions can be marked as private. Also, while a class can [extend](#) (or subclass) only one abstract class, it can implement many interfaces.

A good rule of thumb is this: When you need a category of behavior or properties that objects have in common that does not fit using inheritance, use an interface. If, on the other hand, inheritance makes sense – but you do not want a concrete parent class – then an abstract class may make sense. (And if you want to be able to construct your parent class, then a regular class is still your best bet.)

Combat in NyetHack

Adding combat to NyetHack will put to use all that you have learned about object-oriented programming.

Some rooms in NyetHack – the long corridor and dungeon – will contain a monster for your hero to vanquish in the most graphic way that you know how: by nullifying it.

Create a new kind of room called a **MonsterRoom** to define a room capable of holding a monster. Add a **monster** property of nullable type **Monster?** to the new **MonsterRoom** class and initialize it by assigning it a **Goblin**. Also, override **Room**'s description to let the player know whether the room has a monster to fight.

Listing 17.9 Defining rooms with monsters (Room.kt)

```
open class Room(val name: String) {
    protected open val status = "Calm"

    open fun description() = "$name (Currently: $status)"

    open fun enterRoom() {
        narrate("There is nothing to do here")
    }
}

open class MonsterRoom(
    name: String,
    var monster: Monster? = Goblin()
) : Room(name) {

    override fun description() =
        super.description() + " (Creature: ${monster?.description ?: "None"})"

    override fun enterRoom() {
        if (monster == null) {
            super.enterRoom()
        } else {
            narrate("Danger is lurking in this room")
        }
    }
}
```

Notice the calls to `super.description()` and `super.enterRoom()`. The `super` keyword is used to call into the superclass and access the non-overridden behavior of a function or property. Here, you are using it to call **Room**'s **description** and **enterRoom** implementations and build on top of them when implementing the **MonsterRoom**.

Your new **MonsterRoom** keeps track of any lurking fiends in a property called `monster`. If this property is null, then the monster has been bested. Otherwise, your hero still has a foe to defeat.

You initialized `monster`, a property of type **Monster?**, with an object of type **Goblin**. A monster room can host any subclass of **Monster**, and **Goblin** is a subclass of **Monster** – this is polymorphism at work. If you were to create another class that subclasses **Monster**, then it could also be used in a monster room.

To make use of your new **MonsterRoom** type, you need to strategically change some rooms in NyetHack's map. Update the `worldMap` property in **Game** to add foes in the sketchier parts of town.

Listing 17.10 Placing monsters (NyetHack.kt)

```
...
object Game {
    private val worldMap = listOf(
        listOf(TownSquare(), Tavern(), Room("Back Room")),
        listOf(MonsterRoom("A Long Corridor"), Room("A Generic Room")),
        listOf(MonsterRoom("The Dungeon"))
    )
    ...
}
```

With the monsters in place, add a function called **fight** to **Game**:

Listing 17.11 Defining the fight function (NyetHack.kt)

```
...
object Game {
    ...
    fun move(direction: Direction) {
        ...
    }

    fun fight() {
        val monsterRoom = currentRoom as? MonsterRoom
        val currentMonster = monsterRoom?.monster
        if (currentMonster == null) {
            narrate("There's nothing to fight here")
            return
        }

        while (player.healthPoints > 0 && currentMonster.healthPoints > 0) {
            player.attack(currentMonster)
            if (currentMonster.healthPoints > 0) {
                currentMonster.attack(player)
            }
            Thread.sleep(1000)
        }

        if (player.healthPoints <= 0) {
            narrate("You have been defeated! Thanks for playing")
            exitProcess(0)
        } else {
            narrate("${currentMonster.name} has been defeated")
            monsterRoom.monster = null
        }
    }
}
```



```

    }
    private class GameInput(arg: String?) {
        ...
    }
}

```

fight first checks to see whether the current room’s `monster` is null. If it is, then there is nothing to fight, and a corresponding message is returned. If there is a monster to fight, then – as long as the player and the monster still have at least 1 health point – a round of combat is performed.

If the player’s `healthPoints` value reaches 0, then the game ends, which you achieve with a call to **exitProcess**. **exitProcess** is a Kotlin standard library function that terminates the running instance of your program. (It is available in Kotlin/JVM and Kotlin/Native, but not Kotlin/JS.) To access this function, you will have to import **kotlin.system.exitProcess**.

If the monster’s `healthPoints` value reaches 0, then the monster is nullified in dramatic fashion.

In each round of combat, you call the **attack** function on the monster and on the player. The same **attack** function can be called on both **Monster** and **Player** because they both implement the `Fightable` interface.

Test your new combat system by adding a “fight” command to **GameInput** that calls the **fight** function.

Listing 17.12 Adding the fight command (NyetHack.kt)

```

...
object Game {
    ...
    private class GameInput(arg: String?) {
        private val input = arg ?: ""
        val command = input.split(" ")[0]
        val argument = input.split(" ").getOrNull(1) { "" }

        fun processCommand() = when (command.lowercase()) {
            "fight" -> fight()
            "move" -> {
                val direction = Direction.values()
                    .firstOrNull { it.name.equals(argument, ignoreCase = true) }
                if (direction != null) {
                    move(direction)
                } else {
                    narrate("I don't know what direction that is")
                }
            }
            else -> narrate("I'm not sure what you're trying to do")
        }
    }
}
}

```

Run `NyetHack.kt`. Try moving south into the long corridor and using the “fight” command to engage in combat. The randomness that you introduced in the **attack** function on the `Fightable` interface means that you will have a different experience each time you walk into a new room and pick a fight.

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: move south
The hero moves South
Madrigal of Neversummer, The Renowned Hero, is in A Long Corridor
(Currently: Calm) (Creature: A nasty-looking goblin)
Danger is lurking in this room
> Enter your command: fight
Madrigal deals 9 to Goblin
Goblin deals 13 to Madrigal
Madrigal deals 8 to Goblin
Goblin deals 7 to Madrigal
Madrigal deals 5 to Goblin
Goblin deals 12 to Madrigal
Madrigal deals 6 to Goblin
Goblin deals 6 to Madrigal
Madrigal deals 11 to Goblin
Goblin has been defeated
Madrigal of Neversummer, The Renowned Hero, is in A Long Corridor
(Currently: Calm) (Creature: None)
There is nothing to do here
> Enter your command:

```

In this chapter, you used interfaces to define what a creature (or player) needs in order to engage in combat, and you used abstract classes to create a base class for all monsters in the world of `NyetHack`. These tools will help you create relationships that focus on what a class can do rather than how it does it.

Many of the object-oriented concepts that you have learned about in the past several chapters return to this common goal: Leveraging the tools of the Kotlin framework to create scalable codebases that only expose what they need to and encapsulate the rest.

You have accomplished quite a lot in your journey with `NyetHack`: You laid a foundation of conditionals and functions, defined your own classes so you could represent objects in the world, built a game loop to take input from the player, and even built out a world to explore with monsters to defeat. Congratulations!

In the next chapter, you will learn about generics, a feature that allows you to specify classes that work with many types.

Challenge: Additional Monsters

Right now, you have defined the framework for a diverse set of combatants in NyetHack, but have only placed Goblins in your player's path. Flesh out the world of NyetHack by defining new monsters and randomly placing them in your rooms to make each game of NyetHack a unique combat experience.

For inspiration, try adding classes like **Draugr**, **Werewolf**, and **Dragon**. Think about how much health and what dice rolls each of these creatures should have, given its strength. (The exact numbers are up to you, but a dragon should likely have much more health than a goblin.) When placing monsters, try incorporating rarity in your random generation. For example, dragons in NyetHack are mythical and should not appear often in places like the long corridor adjacent to the town square.

You may need to add more rooms to NyetHack to get a comprehensive monster tour and to test out the scenario where the brave hero loses in battle.

Part V

Advanced Kotlin

In the previous parts of this book, you learned about Kotlin's basic syntax, functional programming, and object-oriented programming. Kotlin has several other tools up its sleeve that both enable and build on these ideas. This section of the book focuses on three topics: generics, extension functions and properties, and coroutines.

18

Generics

You learned in [Chapter 9](#) that a list can hold any type – integers, strings, or even new types that you have defined:

```
val listOfInts: List<Int> = listOf(1, 2, 3)
val listOfStrings: List<String> = listOf("string one", "string two")
val listOfRooms: List<Room> = listOf(Room(), TownSquare())
```

Lists can hold any type because of [generics](#), a type system feature that allows both functions and types to work with types that are not yet known to you or the compiler. Generics greatly expand the reusability of your class definitions, because they allow your definitions to work with many types.

In this chapter, you will create your own generic classes and functions that work with generic type parameters. You will be adding a class called **LootBox**. Each room in NyetHack will be given a loot box, which will hold treasures for the hero to collect.

Defining Generic Types

A [generic type](#) is a class that accepts an input of any type in its constructor (though there can be limits on the type, as you will see later in this chapter). You will begin by defining your own generic type.

In NyetHack, create a new Kotlin file called `Loot.kt`. Within your new file, define a **LootBox** class that specifies a [generic type parameter](#) for use with its contents and contains a private property called `contents` that is assigned the item.

Listing 18.1 Creating a generic class (Loot.kt)

```
class LootBox<T>{var contents: T}
```

You define the **LootBox** class and make it generic by specifying a generic type parameter for use with the class, written as **T** and specified within diamond braces (< >) like other type parameters. The generic type parameter, **T**, is a placeholder for the item’s type.

The generic type parameter is often represented like this, with the single letter **T** (short for “type”), though any letter or word can be used. Some conventions are to use **K** as a shorthand for “key,” **V** for “value” (as in a key-value pair), **E** for “element,” and **R** for “result.” If you need to use multiple generics that do not fit these conventions, it is typical to use **T**, **U**, and **V**. You are also free to use full words for your generic type names, if you prefer.

The **LootBox** class accepts an item of any type as a primary constructor value (`var contents: T`) and stores it in a property using the shorthand syntax you saw in [Chapter 14](#).

Your loot boxes will also need to have some items that they can contain. Create three kinds of loot to be placed into loot boxes: fedoras, gemstones, and keys (which might let the player into exciting areas that they otherwise could not enter).

Listing 18.2 Defining loot (Loot.kt)

```
class LootBox<T>{var contents: T}
```

```
class Fedora(  
    val name: String,  
    val value: Int  
)
```

```
class Gemstones(  
    val value: Int  
)
```

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

With your loot in place, it is time to put the new **LootBox** class to the test. Create a couple loot boxes in your **main** function.

Listing 18.3 Creating loot boxes (NyetHack.kt)

```
...  
fun main() {  
    narrate("Welcome to NyetHack!")  
    val playerName = promptHeroName()  
    player = Player(playerName)  
    // changeNarratorMood()  
  
    val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))  
    val lootBoxTwo: LootBox<Gemstones> = LootBox(Gemstones(150))  
  
    Game.play()  
}  
...
```

Because you made the **LootBox** class generic, you are able to use just one class definition to support different kinds of loot boxes: ones that hold fedoras, ones that hold gemstones, and so on.

Notice the type signature for each **LootBox** variable:

```
val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))  
val lootBoxTwo: LootBox<Gemstones> = LootBox(Gemstones(150))
```

The diamond braces on the type of both variables show what type of loot a particular **LootBox** instance is capable of holding.

Generic types, like other types in Kotlin, support type inference. We have included the type explicitly for illustration, but it could have been omitted since each variable is initialized with a value. In your own code, you typically will omit the type information when it is not needed. If you wanted to, you could remove the type here. Your loot box declarations would then look like this:

```
val lootBoxOne = LootBox(Fedora("a generic-looking fedora", 15))  
val lootBoxTwo = LootBox(Gemstones(150))
```

Generic Functions

Generic type parameters also work with functions. That is good news, since there is currently no way for a player to take the loot out of a loot box.

Time to fix that. Add a function that lets a player take the loot if they have not already collected it. Keep track of whether the loot has been collected by marking whether the loot box has been opened, using an `isOpen` property.

Listing 18.4 Adding a `takeLoot` function (`Loot.kt`)

```
class LootBox<T>(var contents: T) {
    var isOpen = false
    private set

    fun takeLoot(): T? {
        return contents.takeIf { !isOpen }
            .also { isOpen = true }
    }
}
...
```

Here you define a generic function, **takeLoot**, that returns **T?** – a nullable version of the generic type parameter specified on the **LootBox** class, which is a placeholder for the type of the item. If **takeLoot** were defined outside **LootBox**, type **T** would not be available, because **T** is tied to the **LootBox** class definition. However, a function does not require a class to use a generic type parameter, as you will see in the next section.

Try fetching the contents of `lootBoxOne` in the **main** function using the new **takeLoot** function. In fact, try fetching the contents twice:

Listing 18.5 Testing the generic `takeLoot` function (`NyetHack.kt`)

```
...
fun main() {
    ...
    val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))
    val lootBoxTwo: LootBox<Gemstones> = LootBox(Gemstones(150))

    repeat(2) {
        narrate(
            lootBoxOne.takeLoot()?.let {
                "The hero retrieves ${it.name} from the box"
            } ?: "The box is empty"
        )
    }
    ...
}
...
```

You use the scope function **let** (which you learned about in [Chapter 12](#)) to narrate the hero’s experience of opening `lootBoxOne`. Recall that **let** provides an argument (accessed with the `it` identifier) with the value of the receiver it was called on. Because type **T** is known for `lootBoxOne` (you explicitly declare it as **Fedora**), the return result of **takeLoot** – and `it` – is known to be a **Fedora**.

Run `NyetHack`. Your output should reflect that the hero successfully removes loot from the box and then finds the box empty:

```
Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
The hero retrieves a generic-looking fedora from the box
The box is empty
Welcome, adventurer
...
```


Generic Constraints

At this moment, it is possible to put anything inside your loot box. This is not ideal, because you do not want your hero to be rewarded with a **Monster** via a loot box. You can ensure that loot boxes are only used to hold loot by specifying a generic type constraint.

Start by introducing an abstract class called **Loot** and an interface called `Sellable` in `Loot.kt`. Then make your loot classes extend from them, as shown in [Listing 18.6](#). (Keys will not be sellable, because nobody would want to purchase a random key.)

Listing 18.6 Adding superclasses (Loot.kt)

```
...
abstract class Loot {
    abstract val name: String
}

interface Sellable {
    val value: Int
}

class Fedora(
    override val name: String,
    override val value: Int
) : Loot(), Sellable

class Gemstones(
    override val value: Int
) : Loot(), Sellable {
    override val name = "sack of gemstones worth $value gold"
}

class Key(
    override val name: String
) : Loot()
```

Now, add a generic type constraint to **LootBox**'s generic type parameter to allow only descendants of the **Loot** class to be used with **LootBox**:

Listing 18.7 Constraining the generic parameter to only Loot (Loot.kt)

```
class LootBox<T : Loot>(var contents: T) {
    ...
}
```

If you do not specify a generic constraint for a generic type, Kotlin will implicitly constrain it to **Any?**, indicating that any type (either nullable or non-nullable) can be used with the generic class. Here, you add a constraint to the generic type **T**, specified as `: Loot`. Now, only items that are descendants of the **Loot** class can be added to the loot box.

Run NyetHack to confirm that your code has no errors and prints the same output as before.

You might be wondering, “Why is **T** still needed here? Why not just use the type **Loot**?” By using **T**, **LootBox** allows you to access a specific kind of **Loot** while allowing the contents to be any kind of **Loot**. So, rather than the **LootBox** containing **Loot**, the **LootBox** can contain a **Fedora** – and the **Fedora**'s specific type is tracked with **T**.

If you used **Loot** for the type, that would constrain **LootBox** to accept descendants of **Loot**, but it would also discard the information that a **Fedora** was in the box. Using **Loot** for the type, for example, this code would not compile:

```
val lootBox: LootBox<Loot> = LootBox(Fedora("a dazzling fuchsia fedora", 15))
val fedora: Fedora = lootBox.contents // Type mismatch. Required Fedora,
// Found Loot
```

It would no longer be possible to see that the **LootBox** contained anything other than **Loot**. By using a type constraint, it is possible to constrain the contents to **Loot** while preserving the specific subtype of the loot in the box.

You can also define more complicated constraints for your generic types. At some point, your hero will likely want to exchange their gemstones and fedoras for cold hard cash. Introduce a new class called **DropOffBox**. Drop-off boxes will be automated collection bins that exchange valuables for currency – in exchange for a hefty 30% convenience fee.

Listing 18.8 Using multiple generic constraints (Loot.kt)

```
class LootBox<T : Loot>(var contents: T) {
    ...
}

class DropOffBox<T> where T : Loot, T : Sellable {
    fun sellLoot(sellableLoot: T): Int {
        return (sellableLoot.value * 0.7).toInt()
    }
}

abstract class Loot {
    abstract val name: String
}

interface Sellable {
```

```
    val value: Int
  }
  ...
```

This new code uses the `where` keyword to specify the generic constraints of type **T**. The constraint you specified indicates that **T** must extend from **Loot** *and* implement `Sellable`. This constraint restricts the drop-off box from accepting keys (which are worthless to all but their owner and individuals willing to do some breaking and entering) and other items in the future that might be sellable but not loot.

Inside the function body, you have access to all the functions and properties declared on both **Loot** and `Sellable`, because type **T** is guaranteed to extend both types.

Later in this chapter, you will be creating an instance of **DropOffBox** in the town square for the hero to exchange their treasures upon returning from their quests. For now, test your new drop-off box in the REPL to ensure that it is exchanging loot as expected, and only if it is sellable. (You may need to reload the REPL with the Build and restart button to its left.)

Listing 18.9 Using a class with multiple constraints (REPL)

```
import com.bignerdranch.nyethack.*

val hatDropOffBox = DropOffBox<Fedora>()
hatDropOffBox.sellLoot(Fedora("a sequin-covered fedora", 20))
14

hatDropOffBox.sellLoot(Gemstones(100))
error: type mismatch: inferred type is Gemstones but Fedora was expected
hatDropOffBox.sellLoot(Gemstones(100))
                        ^
```


in and out

To further customize your generic type parameters, Kotlin provides the keywords `in` and `out`. To see how they affect your generic classes, try running the following code in the REPL:

Listing 18.10 Attempting to reassign `lootBox` (REPL)

```
var fedoraBox: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))
var lootBox: LootBox<Loot> = LootBox(Gemstones(150))

lootBox = fedoraBox
error: type mismatch: inferred type is LootBox<Fedora> but
      LootBox<Loot> was expected
lootBox = fedoraBox
      ^
```

This result may surprise you. The compiler will not allow you to perform the reassignment of `lootBox` to `fedoraBox`.

It might seem like the assignment should have been possible. **Fedora** is, after all, a descendant of **Loot**, and assigning a variable of the **Loot** type an instance of **Fedora** is possible:

```
var loot: Loot = Fedora("a generic-looking fedora", 15) // No errors
```

To understand why the assignment fails, let's walk through what could happen if it succeeded.

If the compiler allowed you to assign the `fedoraBox` instance to the `lootBox` variable, `lootBox` would then point to `fedoraBox`, and it would be possible to interface with `fedoraBox`'s item as **Loot**, instead of **Fedora** (because of `lootBox`'s type, **LootBox<Loot>**).

For example, gemstones are valid **Loot**, so it would be possible to assign gemstones to `lootBox.contents` (which points to `fedoraBox`).

```
var fedoraBox: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))
var lootBox: LootBox<Loot> = LootBox(Gemstones(150))

lootBox = fedoraBox
lootBox.contents = Gemstones(200)
```

Now, suppose you tried to access `fedoraBox.contents`, expecting a `fedora`:

```
var fedoraBox: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))
var lootBox: LootBox<Loot> = LootBox(Gemstones(150))

lootBox = fedoraBox
lootBox.contents = Gemstones(200)
val myFedora: Fedora = fedoraBox.contents
```

Your program would then be faced with a type mismatch – the type of `fedoraBox.contents` is not **Fedora**, it is **Gemstones**. When your program executes in this hypothetical scenario, it would crash with a **ClassCastException**. This is the problem that arises and the reason the assignment is not allowed by the compiler.

It is also why the `in` and `out` keywords exist.

In the **LootBox** class's definition, add the `out` keyword and change `contents` from a `var` to a `val`:

Listing 18.11 Adding `out` (**Loot.kt**)

```
class LootBox<out T : Loot>(var val contents: T) {
    ...
}
```

Next, try re-evaluating the last code snippet you entered into the REPL (from [Listing 18.10](#)). You will see that the REPL does not output any errors, indicating that the code compiles successfully.

What has changed?

There are two roles a generic parameter can be assigned: [producer](#) or [consumer](#). The role of producer means that a generic parameter will be readable (but not writable), and consumer means the generic parameter will be writable (but not readable).

When you added the `out` keyword to `LootBox<out T>`, you specified that the generic would act as a producer. That meant that defining `contents` with the `var` keyword was no longer permitted, since that would cause **LootBox** to be both a producer *and* a consumer of type **T**.

Recall that when your `contents` variable could be reassigned, there were scenarios where its type could change and cause unexpected errors when taking loot out of the loot box. By making the generic a producer, you assure the compiler that this dilemma is no longer a possibility: Because the generic parameter is a producer, never a consumer, the `contents` variable will never change.

Kotlin now permits the assignment of `fedoraBox` to `lootBox`, because it is safe to do so: `lootBox`'s `contents` now has type **Fedora**, not **Loot**, and cannot be changed. Kotlin will only permit this kind of casting

when the type is marked with `out`. And the compiler will enforce the restriction that any type marked `out` can only appear as a return type, not an input.

By the way, **Lists** are also producers. In Kotlin’s definition for **List**, the generic type parameter is marked with the `out` keyword:

```
public interface List<out E> : Collection<E>
```

(**MutableList**, however, is neither a producer nor a consumer. It accepts data as inputs in addition to outputting data in the collection, meaning that you would not be able to safely cast a **MutableList<Fedora>** to a **MutableList<Loot>**.)

Your **DropOffBox**, meanwhile, is a consumer: It accepts values of its generic type and does not output them.

You can mark the generic type parameter for **DropOffBox** with the `in` keyword to have the opposite effect on casting instances of a **DropOffBox**. Instead of being allowed to cast `DropOffBox<Fedora>` to `DropOffBox<Loot>`, you would be allowed to assign `DropOffBox<Loot>` to `DropOffBox<Fedora>` – but not vice versa. As an additional requirement, generic types marked with the `in` keyword cannot be stored in properties, since reading from the property is considered outputting the value, violating the consumer rule.

Update **DropOffBox** to use the `in` keyword.

Listing 18.12 Marking DropOffBox with `in` (Loot.kt)

```
...
class DropOffBox<in T> where T : Loot, T : Sellable {
    fun sellLoot(sellableLoot: T): Int {
        return (sellableLoot.value * 0.7).toInt()
    }
}
...
```

Before you can make use of your new casting abilities, you will also need to declare some more loot types. Create a new abstract class called **Hat**, which will serve as a base class for NyetHack’s most important fashion statements. Next, make **Fedora** extend from **Hat** and introduce a new **Fez** type to add diversity to your headwear.

Listing 18.13 Adding more hats (Loot.kt)

```
...
abstract class Hat : Loot(), Sellable

class Fedora(
    override val name: String,
    override val value: Int
) : Loot(), Sellable Hat()

class Fez(
    override val name: String,
    override val value: Int
) : Hat()
...
```

With the increased hat diversity, Kronstadt has attracted the attention of a traveling hat salesman. This salesman will utilize the drop-off box system.

Initially, the salesman will accept all kinds of hats. If the hat marketplace suddenly becomes overwhelmed with fezzes, the salesman could restrict the drop-off box to only accept fedoras. With the help of the `in` keyword, it is now possible to model this scenario exactly, because it is the same box being used for all hats.

Try out the following code in the REPL to pay a visit to the salesman:

Listing 18.14 Leveraging `in` for casting (REPL)

```
import com.bignerdranch.nyethack.*

val hatDropOffBox: DropOffBox<Hat> = DropOffBox()
val fedoraDropOffBox: DropOffBox<Fedora> = hatDropOffBox

fedoraDropOffBox.sellLoot(Fedora("one-of-a-kind fedora", 1000))
700
```

This assignment is possible because the compiler can be certain you would never be able to produce a **Hat** from a **DropOffBox** for **Fedoras** – avoiding the possibility of class cast exceptions. Because hats never leave the drop-off box, the compiler is able to reason that the assignment you have made is a safe one.

By the way, you may hear the terms [covariance](#) and [contravariance](#) used to describe what `out` and `in` do. We think these terms lack the commonsense clarity of `in` and `out`, so we avoid them. We mention them here because you may encounter them elsewhere, so now you know: If you hear “covariance,” think “out,” and if you hear “contravariance,” think “in.”

Adding Loot to NyetHack

With the loot scaffolding in place, you are ready to scatter precious items throughout NyetHack for your player to collect and sell. Start out by giving the player pockets to store their valuables.

Listing 18.15 Adding an inventory (Player.kt)

```
class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    override var healthPoints: Int,
    val isImmortal: Boolean
) : Fightable {
    ...
    val prophecy by lazy {
        ...
    }

    val inventory = mutableListOf<Loot>()

    var gold = 0
    ...
}
```

Next, add a companion object to **LootBox** and create a **random** function to randomly generate a variety of loot boxes for the player to discover.

Listing 18.16 Randomizing loot (Loot.kt)

```
class LootBox<out T : Loot>(val contents: T) {
    var isOpen = false
    private set

    fun takeLoot(): T? {
        return contents.takeIf { !isOpen }
            .also { isOpen = true }
    }

    companion object {
        fun random(): LootBox<Loot> = LootBox(
            contents = when (Random.nextInt(1..100)) {
                in 1..5 -> Fez("fez of immaculate style", 150)
                in 6..10 -> Fedora("fedora of knowledge", 125)
                in 11..15 -> Fedora("stunning teal fedora", 75)
                in 16..30 -> Fez("ordinary fez", 15)
                in 31..50 -> Fedora("ordinary fedora", 10)
                else -> Gemstones(Random.nextInt(50..100))
            }
        )
    }
}
```

(Remember to import both `kotlin.random.Random` and `kotlin.random.nextInt`.)

This function creates random loot boxes with gemstones or valuable hats to acquire.

With your loot generation in place, you can now add a loot box to each room in NyetHack. Do this by adding an open property to the **Room** class.

Listing 18.17 Adding loot boxes to Room (Room.kt)

```
open class Room(val name: String) {
    ...
    protected open val status = "Calm"
    open val lootBox: LootBox<Loot> = LootBox.random()
    ...
}
```

To allow rooms to provide the player with all kinds of loot, you declare the type as **Loot** – the most generic kind of item that can be kept in a loot box. But even though **Room** specifies that its `lootBox` can contain any kind of loot, you can deviate from this in subclasses, because you marked the property as open.

Suppose the tavern should always have a loot box containing a key that will become useful later. Override the `lootBox` property of **Tavern**:

Listing 18.18 Overriding a generic property (Tavern.kt)

```
...
class Tavern : Room(TAVERN_NAME) {
    ...
    override val status = "Busy"

    override val lootBox: LootBox<Key> =
        LootBox(Key("key to Nogartse's evil lair"))
    ...
}
...
```

This new property is declared as a **LootBox<Key>**, deviating from the parent class’s type of **LootBox<Loot>**. This is allowed because you declared **LootBox**’s generic type **T** as out. The compiler can safely cast **Key** to **Loot** (and, by extension, **LootBox<Key>** to **LootBox<Loot>**), so this override is allowed. Now, if you read the loot box from an instance of **Tavern**, the compiler can see that you will be given a **Key** – so this code would compile and run successfully:

```
val tavern = Tavern()
val key: Key? = tavern.lootBox.takeLoot()
```

With your loot boxes placed and your hero’s pockets ready to be filled, you can implement a new “take loot”

command. While you are doing that, delete the loot boxes from **main**, since the boxes have now been placed in NyetHack’s rooms.

Listing 18.19 Implementing the take loot command (NyetHack.kt)

```
...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()

    val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic looking fedora", 15))
    val lootBoxTwo: LootBox<Gemstones> = LootBox(Gemstones(150))

    repeat(2) {
        narrate{
            lootBoxOne.takeLoot()?.let {
                "The hero retrieves ${it.name} from the box"
            } ?: "The box is empty"
        }
    }

    Game.play()
}
...
object Game {
    ...
    fun takeLoot() {
        val loot = currentRoom.lootBox.takeLoot()
        if (loot == null) {
            narrate("${player.name} approaches the loot box, but it is empty")
        } else {
            narrate("${player.name} now has a ${loot.name}")
            player.inventory += loot
        }
    }

    private class GameInput(arg: String?) {
        ...
        fun processCommand() = when (command.lowercase()) {
            "fight" -> fight()
            "move" -> ...
            "take" -> {
                if (argument.equals("loot", ignoreCase = true)) {
                    takeLoot()
                } else {
                    narrate("I don't know what you're trying to take")
                }
            }
            else -> narrate("I'm not sure what you're trying to do")
        }
        ...
    }
}
```

Take your new looting command out for a spin in the various rooms of NyetHack. The tavern should always grant you the key to Nogartse’s evil lair, but all other rooms will contain random loot. Your output should look something like this:

```
Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:take loot
Madrigal now has a sack of gemstones worth 57 gold
...
> Enter your command:
```

Now that your hero is able to collect loot, you can place drop-off boxes to allow the player to sell valuables for in-game currency. Place two drop-off boxes in the town square: One will collect hats and the other will collect gemstones. Then, create a new function on **TownSquare** called **sellLoot**, which will pick the correct box to sell the loot at.

Listing 18.20 Placing drop-off boxes (TownSquare.kt)

```
open class TownSquare : Room("The Town Square") {
    override val status = "Bustling"
    private var bellSound = "GWONG"
    val hatDropOffBox = DropOffBox<Hat>()
    val gemDropOffBox = DropOffBox<Gemstones>()

    final override fun enterRoom() {
        narrate("The villagers rally and cheer as the hero enters")
        ringBell()
    }

    fun ringBell() {
        narrate("The bell tower announces the hero's presence: $bellSound")
    }

    fun <T> sellLoot(
        loot: T
    ): Int where T : Loot, T : Sellable {
        return when (loot) {
            is Hat -> hatDropOffBox.sellLoot(loot)
            is Gemstones -> gemDropOffBox.sellLoot(loot)
            else -> 0
        }
    }
}
```

To utilize your new drop-off boxes, create one more command – “sell loot”:

Listing 18.21 Selling loot (NyetHack.kt)

```
...
object Game {
    ...
    fun sellLoot() {
        when (val currentRoom = currentRoom) {
            is TownSquare -> {
                player.inventory.forEach { item ->
                    if (item is Sellable) {
                        val sellPrice = currentRoom.sellLoot(item)
                        narrate("Sold ${item.name} for $sellPrice gold")
                    }
                }
            }
        }
    }
}
```

```

        player.gold += sellPrice
    } else {
        narrate("Your ${item.name} can't be sold")
    }
}
player.inventory.removeAll { it is Sellable }
}
else -> narrate("You cannot sell anything here")
}
}

private class GameInput(arg: String?) {
    ...
    fun processCommand() = when (command.lowercase()) {
        ...
        "take" -> {
            if (argument.equals("loot", ignoreCase = true)) {
                takeLoot()
            } else {
                narrate("I don't know what you're trying to take")
            }
        }
        "sell" -> {
            if (argument.equals("loot", ignoreCase = true)) {
                sellLoot()
            } else {
                narrate("I don't know what you're trying to sell")
            }
        }
        else -> narrate("I'm not sure what you're trying to do")
    }
    ...
}
}

```

You now have two functions called **sellLoot: TownSquare** has a **sellLoot** function to sell a given piece of loot at an appropriate drop-off container, and **Game** has another **sellLoot** function that attempts to sell the player’s loot based on what room the player is in. If the player is in the town square, **Game**’s **sellLoot** function calls **TownSquare**’s **sellLoot** function and then removes all sellable items from the player’s inventory. Otherwise, loot cannot be sold in the user’s current location.

Your loot system is now complete. Test it out by visiting various rooms in NyetHack, collecting their valuables, and then exchanging them for money. Your output should look something like this:

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:take loot
Madrigal now has a sack of gemstones worth 70 gold
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:move east
The hero moves East
Madrigal of Neversummer, The Renowned Hero, is in Taernyl's Folly
(Currently: Busy)
...
> Enter your command:take loot
Madrigal now has a key to Nogartse's evil lair
...
> Enter your command:move west
The hero moves West
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:sell loot
Sold sack of gemstones worth 70 gold for 49 gold
Your key to Nogartse's evil lair can't be sold
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:

```

In this chapter, you have learned how to use generics to expand the capabilities of Kotlin’s classes. You have also seen type constraints and how the `in` and `out` keywords can be used to define the producer and consumer roles for the generic parameter.

In the next chapter, you will learn about extensions, which allow you to share functions and properties without using inheritance. You will use them to improve NyetHack’s codebase.

For the More Curious: The reified Keyword

There are cases where it is useful to know the specific type that is used for a generic parameter. The `reified` keyword allows you to check a generic parameter's type.

Suppose that Madrigal is on a quest to collect loot of a specific type. For example, she may want to collect every hat she can get her hands on, ignoring all other types of loot. Here is a **takeLootOfType** function that attempts to capture that logic:

```
class LootBox<out T : Loot>(val contents: T) {
    var isOpen = false
    private set

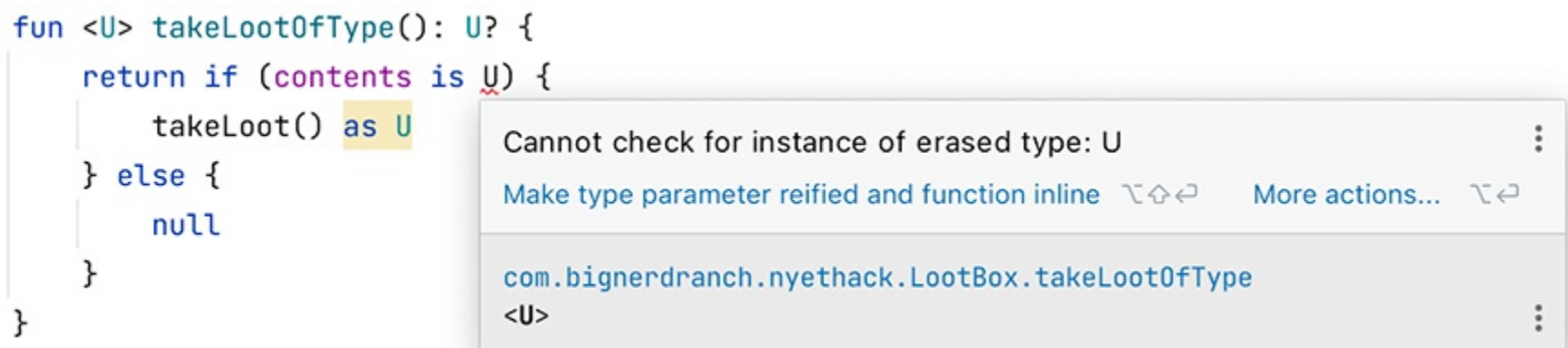
    fun takeLoot(): T? {
        return contents.takeIf { !isOpen }
            .also { isOpen = true }
    }

    fun <U> takeLootOfType(): U? {
        return if (contents is U) {
            takeLoot() as U
        } else {
            null
        }
    }
}

val lootBox = LootBox.random()
val loot = lootBox.takeLootOfType<Hat>()
```

If you typed this in, you would find that it does not work. IntelliJ would flag the type parameter **U** with an error ([Figure 18.1](#)):

Figure 18.1 Cannot check for instance of erased type: U



Information about generic types is usually only available at compile time. When you compile your code, Kotlin checks that your generic types match your usage of the class, then omits the generic type information from your compiled code. In practice, this means that **LootBox<Hat>** and **LootBox<Gemstones>** both compile into **LootBox**, with no additional type information. This is known as *type erasure* – meaning the type information for a generic type is not available at runtime.

Because your `is` check happens at runtime, your compiled program will not have enough information to know what type **U** is. However, Kotlin provides a way around this using the `reified` keyword. When you reify a generic type, you tell the compiler that you want the generic type to be available at runtime. To use the `reified` keyword, you must also inline the function.

To make the same **takeLootOfType** function reified – and to address the compiler error from before – you can add the `inline` and `reified` keywords, like this:

```
inline fun <reified U> takeLootOfType(): U? {
    return if (contents is U) {
        takeLoot() as U
    } else {
        null
    }
}
```

Now the type check `contents is U` is possible, because the type information is reified. The Kotlin compiler will preserve your type information by inlining the function and replacing the reified type with the actual type being used. Because you need to mark the function as `inline`, you are subject to the same limitations and caveats for other inlined functions that you saw in [Chapter 8](#).

Using the `reified` keyword allows you to inspect the type of a generic parameter at runtime safely and efficiently.

Extensions

[Extensions](#) allow you to add functionality to a type without directly modifying the type's definition. You can use extensions with your own types and also types you do not control, like **List**, **String**, and other types from the Kotlin standard library.

Extensions are an alternative to the sharing behavior of inheritance. They are a good fit for adding functionality to a type when you do not control the definition of the class or when a class is not marked with the `open` keyword and therefore is ineligible for subclassing.

The Kotlin standard library frequently uses extensions. For example, the scope functions that you learned about in [Chapter 12](#) are defined as extensions, and you will look at some examples of their declarations in this chapter.

Defining Extension Functions

Your first extension allows you to add a specified amount of enthusiasm to any **String**. Define it in a new file called `Extensions.kt` in `NyetHack`:

Listing 19.1 Adding an extension to `String` (`Extensions.kt`)

```
fun String.addEnthusiasm(enthusiasmLevel: Int = 1) =
    this + "!".repeat(enthusiasmLevel)
```

Extension functions are defined in the same way as other functions, with one major difference: When you specify an extension function, you also specify the type the extension adds functionality to, known as the [receiver type](#). (Recall from [Chapter 12](#) that the subject of an extension is called a “receiver.”) For the **addEnthusiasm** function, the receiver type you specified is **String**.

addEnthusiasm’s function body is a single-expression function that returns a new string: the contents of `this` plus some number of exclamation points, based on the argument passed to `enthusiasmLevel` (1, if the default value is used). The `this` keyword refers to the receiver instance the extension function was called on (a **String** instance, in this case).

Now, you can invoke the **addEnthusiasm** function on any instance of **String**. The narrator sits at the edge of their seat whenever a hero engages in combat. Increase their enthusiasm over the course of a battle using your new extension function.

Listing 19.2 Calling the new extension on a `String` receiver instance (`NyetHack.kt`)

```
...
object Game {
    ...
    fun fight() {
        val monsterRoom = currentRoom as? MonsterRoom
        val currentMonster = monsterRoom?.monster
        if (currentMonster == null) {
            narrate("There's nothing to fight here")
            return
        }

        var combatRound = 0
        val previousNarrationModifier = narrationModifier
        narrationModifier = { it.addEnthusiasm(enthusiasmLevel = combatRound) }

        while (player.healthPoints > 0 && currentMonster.healthPoints > 0) {
            combatRound++

            player.attack(currentMonster)
            if (currentMonster.healthPoints > 0) {
                currentMonster.attack(player)
            }
            Thread.sleep(1000)
        }
        narrationModifier = previousNarrationModifier

        if (player.healthPoints <= 0) {
            narrate("You have been defeated! Thanks for playing.")
            exitProcess(0)
        } else {
            narrate("${currentMonster.name} has been defeated!")
            monsterRoom.monster = null
        }
    }
    ...
}
```

Run `NyetHack` and fight a monster to see that your extension function adds exclamation points to your narrator’s messages, as expected.

```
...
Madrigal of Neversummer, The Renowned Hero, is in A Long Corridor
(Currently: Calm) (Creature: A nasty-looking goblin)
Danger is lurking in this room
> Enter your command:fight
Madrigal deals 6 to Goblin!
Goblin deals 8 to Madrigal!
Madrigal deals 5 to Goblin!!
Goblin deals 8 to Madrigal!!
Madrigal deals 2 to Goblin!!!
Goblin deals 8 to Madrigal!!!
Madrigal deals 6 to Goblin!!!!
Goblin deals 6 to Madrigal!!!!
Madrigal deals 5 to Goblin!!!!
Goblin deals 8 to Madrigal!!!!
Madrigal deals 4 to Goblin!!!!
Goblin deals 4 to Madrigal!!!!
Madrigal deals 6 to Goblin!!!!
Goblin has been defeated
Madrigal of Neversummer, The Renowned Hero, is in A Long Corridor
(Currently: Calm) (Creature: None)
There is nothing to do here
> Enter your command:
```

Could you have subclassed **String** to add this functionality to **String** instances? In IntelliJ, view **String**’s source definition by pressing the Shift key twice to open the Search Everywhere dialog and then searching for the `String.kt` file, where the type is defined. Its header looks like this:

```
public class String : Comparable<String>, CharSequence {
    ...
}
```

Since there is no open keyword on the **String** class definition, there is no way to subclass **String** to add functionality through inheritance. As we said earlier, extensions are a good option when you want to add functionality to a class you do not control or that is ineligible for subclassing.

Defining an extension on a superclass

Extensions do not rely on inheritance, but they can be combined with inheritance to expand their scope. Try it in `Extensions.kt`: Define an extension on the **Any** class called **print**. Because it is defined on **Any**, it will be directly callable on all types.

Listing 19.3 Extending Any (Extensions.kt)

```
fun String.addEnthusiasm(enthusiasmLevel: Int = 1) =
    this + "!".repeat(enthusiasmLevel)

fun Any.print() {
    println(this)
}
```

Now try out your new **print** function in the REPL:

Listing 19.4 print is available on all subtypes (REPL)

```
import com.bignerdranch.nyethack.*

"Hello from String!".print()
42.print()
Hello from String!
42
```

Generic extension functions

What if you wanted to print the string "Madrigal has left the building" both before and after calling **addEnthusiasm** on it?

First, you would need to make the **print** function chainable. You have seen chained function calls before; functions can be chained if they return their receiver or another object that subsequent functions can be called on.

Update **print** to make it chainable:

Listing 19.5 Making print chainable (Extensions.kt)

```
fun String.addEnthusiasm(enthusiasmLevel: Int = 1) =
    this + "!".repeat(enthusiasmLevel)

fun Any.print(): Any {
    println(this)
    return this
}
```

Now, try calling the **print** function two times: once before **addEnthusiasm** and once after it:

Listing 19.6 Calling print twice (REPL)

```
"Madrigal has left the building".print().addEnthusiasm().print()
error: unresolved reference. None of the following candidates is applicable
      because of receiver type mismatch:
      public fun String.addEnthusiasm(enthusiasmLevel: Int = ...): String defined in
          com.bignerdranch.nyethack
"Madrigal has left the building".print().addEnthusiasm().print()
      ^
```

The code does not compile. The first **print** call was allowed, but **addEnthusiasm** was not. To understand why, take another look at **addEnthusiasm**'s return type:

```
fun Any.print(): Any
```

Because your **print** function returns the **Any** type, the type information of the receiver is lost and you do not get a **String** back. While you could fix this by inserting a cast, it would be better if **print** returned the same type it is called on, like a **String** when it is called on a **String** instance.

To solve this, you can make the **print** function generic and indicate that it returns the same type that it was called on. Update your function to use a generic type as its receiver instead of **Any**:

Listing 19.7 Making print generic (Extensions.kt)

```
...
fun <T> AnyT.print(): AnyT {
    println(this)
    return this
}
```

Now that the extension uses the generic type parameter `T` for the receiver and returns `T` instead of **Any**, the particular type information for the receiver is passed forward in the chain of calls.

Run the code from [Listing 19.6](#) again. This time you will see the following output:

```
Madrigal has left the building
Madrigal has left the building!
```

Your new generic extension function works with any type, and it also maintains the type information. Extensions used with generic types allow you to write functions that have a wide reach across a number of different types in your program.

Extensions on generic types appear throughout the Kotlin standard library. For example, take a look at the

definition for the **let** function:

```
public inline fun <T, R> T.let(block: (T) -> R): R {
    return block(this)
}
```

let is defined as a generic extension function, allowing it to work with all types. It accepts a lambda that takes the receiver as its argument (T) and returns R – some new type that is whatever the lambda returns.

Notice that the `inline` keyword you learned about in [Chapter 8](#) is also used here. The same guidance from before applies: Inlining the extension function if it accepts a lambda reduces the memory overhead required.

Operator extension functions

In [Chapter 16](#), you saw the `operator` keyword and used it to provide an implementation of the plus operator. You can combine extension functions with the `operator` keyword to provide implementations of Kotlin’s operators for types you do not control.

Think back to your **Coordinate** type, which you use to access the current room with this line of code:

```
worldMap.getOrNull(newPosition.y)?.getOrNull(newPosition.x)
```

This is quite a bit of syntax to index into your list. To clean this up, define another extension function to get the right **Room** from the `worldMap` for a given **Coordinate**, like this:

Listing 19.8 Defining an operator extension function (`Extensions.kt`)

```
...
fun <T> T.print(): T {
    println(this)
    return this
}

operator fun List<List<Room>>.get(coordinate: Coordinate) =
    getOrNull(coordinate.y)?.getOrNull(coordinate.x)
```

This lets you use the `get` operator (`[]`) on your `worldMap` to fetch the next room instead of using two `get` calls:

Listing 19.9 Using an operator function defined as an extension (`NyetHack.kt`)

```
...
object Game {
    ...
    fun move(direction: Direction) {
        val newPosition = direction.updateCoordinate(currentPosition)
        val newRoom = worldMap.getOrNull(newPosition.y)?.getOrNull(newPosition.x)
        val newRoom = worldMap[newPosition]
        ...
    }
    ...
}
```

This new syntax makes the intent of your code more obvious and hides the implementation detail of the lookup into the extension function itself. As an added bonus, if another function needs to find a room based on its coordinate, it can now do so without declaring this logic again.

Run `NyetHack` and confirm that you are still able to move between rooms.

Another type of function that you can use with extension functions is an [infix function](#). `infix` is a function modifier that you can use on extension or class functions that have exactly one parameter. When a function is marked as `infix`, it lets you omit the dot (`.`) before the function name and the parentheses around the argument.

You have already used an `infix` function in your code. Take a look at the definition of **to**, which you learned about in [Chapter 10](#):

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

This lets you call the **to** function in either of these ways, making it look almost like an operator in the language itself:

Full syntax `playerName.to(hometown)`

Infix notation `playerName to hometown`

Most developers choose not to declare their own `infix` functions, but you can if you think it improves readability or if you want to make a new function that feels like a first-class citizen in the language itself. To see how this works, create an extension function called **move** to make it easier to alter your **Coordinates**:

Listing 19.10 Declaring an infix extension function (`Extensions.kt`)

```
...
operator fun List<List<Room>>.get(coordinate: Coordinate) =
    getOrNull(coordinate.y)?.getOrNull(coordinate.x)

infix fun Coordinate.move(direction: Direction) =
    direction.updateCoordinate(this)
```

Now, use the extension function – and the `infix` notation – in your **Game** class:

Listing 19.11 Using an infix extension function (NyetHack.kt)

```
...
object Game {
    ...
    fun move(direction: Direction) {
        val newPosition = direction.updateCoordinate(currentPosition)
        val newPosition = currentPosition move direction
        val newRoom = worldMap[newPosition]

        if (newRoom != null) {
            ...
        } else {
            ...
        }
    }
    ...
}
```

Extension Properties

In addition to adding functionality to a type by specifying extension functions, you can also define extension properties. Add another extension to **String** in `Extensions.kt`, this time an extension property that counts a string’s vowels:

Listing 19.12 Adding an extension property (Extensions.kt)

```
fun String.addEnthusiasm(enthusiasmLevel: Int = 1) =
    this + "!".repeat(enthusiasmLevel)

val String.numVowels
    get() = count { it.lowercase() in "aeiou" }
...
```

Try out your new extension property by updating the “Master of Vowels” condition in **Player**:

Listing 19.13 Using an extension property (Player.kt)

```
class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    override var healthPoints: Int,
    val isImmortal: Boolean
) : Fightable {
    ...
    val title: String
        get() = when {
            name.all { it.isDigit() } -> "The Identifiable"
            name.none { it.isLetter() } -> "The Witness Protection Member"
            name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowels"
            name.numVowels > 4 -> "The Master of Vowels"
            else -> "The Renowned Hero"
        }
    ...
}
```

Before you can test this extension property, you need to make another change. In [Chapter 9](#), you commented out the code that asked players to enter a name when starting NyetHack. We mentioned that you would be adding it back in when you neared the end of your work in NyetHack, and now is that time.

Update **promptHeroName** to once again ask players to name their heroes. While you are at it, uncomment the call to **changeNarratorMood** to return your narrator to its mercurial glory.

Listing 19.14 Asking for names (NyetHack.kt)

```
...
fun main() {
    ...
    // changeNarratorMood()

    Game.play()
}

private fun promptHeroName(): String {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Prints the message in yellow
        "\u001b[33;1m$message\u001b[0m"
    }

    //val input = readLine()
    require(input != null && input.isNotEmpty()) {
        "The hero must have a name."
    }

    return input//
    println("Madrigal")
    return "Madrigal"
}
...
```

Run NyetHack. When prompted to enter a name, type “Aurelia” or another vowel-rich name. You will see that your hero is assigned the “Master of Vowels” title, just as before:

```
Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Aurelia
THE NARRATOR BEGINS TO FEEL LOUD!!!
WELCOME, ADVENTURER!!!
AURELIA, A MORTAL, HAS 100 HEALTH POINTS!!!
AURELIA OF NEVERSUMMER, THE MASTER OF VOWELS, IS IN THE TOWN SQUARE
(CURRENTLY: BUSTLING)!!!
THE VILLAGERS RALLY AND CHEER AS THE HERO ENTERS!!!
THE BELL TOWER ANNOUNCES THE HERO'S PRESENCE: GWONG!!!
> Enter your command:
```

Recall from [Chapter 13](#) that class properties (except computed properties) have a backing field where their data is stored and that they are automatically assigned getters and, if needed, setters. Extension properties cannot have a backing field, so they must be computed properties. Every extension property must define **get** and (for vars) **set** functions that compute the value that should be returned by the property to be valid.

For example, this would not be allowed:

```
val String.numberOfWords = 10
error: extension property cannot be initialized because it has no backing field
```

Instead, you could define a valid `numberOfWords` extension property by defining a getter for the `numberOfWords` val.

Extensions on Nullable Types

An extension can also be defined for use with a nullable type. Defining an extension on a nullable type allows you to deal with the possibility of the value being null within the body of the extension function, rather than at the call site.

Add an extension for nullable **Rooms** in `Extensions.kt` that will return a room outside the town of Kronstadt.

Listing 19.15 Adding an extension on a nullable type (`Extensions.kt`)

```
...
infix fun Coordinate.move(direction: Direction) =
    direction.updateCoordinate(this)

fun Room?.orEmptyRoom(name: String = "the middle of nowhere"): Room =
    this ?: Room(name)
```

Use your extension function in the **move** function, letting the player walk outside the bounds of town.

Listing 19.16 Calling an extension on a nullable type (`NyetHack.kt`)

```
...
object Game {
    ...
    fun move(direction: Direction) {
        val newPosition = currentPosition move direction
        val newRoom = worldMap[newPosition].orEmptyRoom()

        if (newRoom != null) {
        narrate("The hero moves ${direction.name}")
        currentPosition = newPosition
        currentRoom = newRoom
        } else {
        narrate("You cannot move ${direction.name}")
    }
}
...
}
```

Run `NyetHack` and try moving to the north. You should see that your hero is not restricted by the world map and is greeted with the vast emptiness surrounding the town. (You should also be able to explore the other rooms in `NyetHack`, as before.)

```
...
Welcome, adventurer?
Madrigal, a mortal, has 100 health points?
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)?
The villagers rally and cheer as the hero enters?
The bell tower announces the hero's presence: GWONG?
> Enter your command: move north
The hero moves North?
Madrigal of Neversummer, The Renowned Hero, is in the middle of nowhere
(Currently: Calm)?
There is nothing to do here?
> Enter your command:
```

Extensions, Under the Hood

An extension function or property is called in the same style as a normal function or property, but it is not directly defined on the class it extends, nor does it rely on inheritance for adding functionality. So how are extensions implemented?

To inspect how an extension works, you can look at the bytecode that the Kotlin compiler generates when you define one and translate it back to Java.

With the text cursor in `Extensions.kt`, open the Kotlin bytecode tool window, either by selecting `Tools → Kotlin → Show Kotlin Bytecode` or by searching for “show Kotlin bytecode” in the Search Everywhere dialog (accessed by pressing the Shift key twice).

In the Kotlin bytecode window, click the `Decompile` button at the top left to open a new tab with the Java representation of the bytecode that was generated from `Extensions.kt`. Find the equivalent bytecode for the **`addEnthusiasm`** extension that you defined for **`String`**:

```
@NotNull
public static final String addEnthusiasm(@NotNull String $this$addEnthusiasm,
                                         int enthusiasmLevel) {
    Intrinsics.checkNotNullParameter($this$addEnthusiasm, "$this$addEnthusiasm");
    return $this$addEnthusiasm +
        StringsKt.repeat((CharSequence) "!", enthusiasmLevel);
}
```

Kotlin extensions are static functions that accept the receiver as their first argument. The compiler substitutes a call of the **`addEnthusiasm`** function. This is how you are able to declare extension functions for every type in Kotlin.

A side effect of this implementation is that extension functions cannot override functions on the base type and cannot access any private properties or functions on the class. If you make an extension with the same signature as a property or function on the class itself, the compiler will favor the definition on the base type over your extension. To see this for yourself, try running this code in the REPL:

Listing 19.17 Resolving extension functions (REPL)

```
val String.length
    get() = -999

"Madrigal has left the building".length
30
```

This expression returns 30, indicating that the built-in `length` property was used instead of the extension. Be careful when declaring extensions to avoid conflicting with the base type. We recommend using a different name entirely, but it is possible to overload a function on a type with an extension function so long as the parameters are different.

Extension Visibility

In this chapter, you did not declare any of your extensions with a visibility modifier, so they are implicitly public. Public extensions are accessible from anywhere in your codebase. At scale, you may find that the classes you extend can become bloated with the number of new functions and properties you have added.

You can avoid this by opting for a regular function rather than an extension, but there are a lot of situations where extensions do a better job of simplifying your code. Another option to prevent extension bloat is to add visibility modifiers to your extensions – much like you can do with regular functions and properties.

Marking an extension as private prohibits use of the extension outside the file it is defined in. Consider the **List<List<Room>>.get** function you defined. You may only want to have access to it for **Game**, since that is where your navigation logic and **worldMap** are defined. To restrict access to this function and to avoid cluttering your IDE's autocomplete recommendations, you could move this function into **Nyethack.kt** and mark it as private.

The rule of thumb is the same for extensions as it is for functions: If the extension will not be used elsewhere, mark it private.

As you may be realizing at this point, extension functions can be declared with almost all the function modifiers that you can use when defining a function inside a class. The **abstract** and **open** keywords are notable exceptions, because it is not possible to override an extension function. Virtually every function could be made an extension function as long as it does not need to access the internals of a class and does not need to be overridden.

Extensions in the Kotlin Standard Library

A large portion of the Kotlin standard library’s functionality is defined via extension functions and extension properties.

For example, take a look at the source code file `Strings.kt` (note: `Strings`, not `String`), by using the Search Everywhere dialog to search for its name:

```
public inline fun CharSequence.trim(predicate: (Char) -> Boolean): CharSequence {
    var startIndex = 0
    var endIndex = length - 1
    var startFound = false

    while (startIndex <= endIndex) {
        val index = if (!startFound) startIndex else endIndex
        val match = predicate(this[index])

        if (!startFound) {
            if (!match)
                startFound = true
            else
                startIndex += 1
        }
        else {
            if (!match)
                break
            else
                endIndex -= 1
        }
    }

    return subSequence(startIndex, endIndex + 1)
}
```

Browse through this standard library file, and you will see that it consists of extensions to the **`String`** type. The excerpt above, for example, defines an extension function **`trim`** that is used to remove characters from a string.

Standard library files that contain extensions to a type are often named in this way, with an `-s` appended to the type name. If you look through the standard library files, you will find other files matching this same naming convention: `Sequences.kt`, `Ranges.kt`, and `Maps.kt` are just some of the files that add functionality to the standard library through extensions to their corresponding type.

Heavy use of extension functions to define core API functionality is one of the ways the Kotlin standard library keeps such a small footprint (approximately 1.4 MB) but packs in so many features. Extensions use space efficiently because they can provide a feature for many types with one definition.

In this chapter, you have learned how extensions provide an alternative to sharing behavior with inheritance. And with that, it is time to say goodbye to NyetHack. Over the past 12 chapters, you have seen much of what Kotlin has to offer, including lambdas, collections, classes, inheritance, generics, and extension functions. In the next chapter, you will be creating a new project to learn about coroutines – Kotlin’s first-class approach to performing asynchronous tasks.

For the More Curious: Function Literals with Receivers

It is possible to use function literals with the extension syntax to powerful effect. To understand what is meant by “function literals with receivers,” take a look at the definition for **apply**, a function you met in [Chapter 12](#):

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

Remember what **apply** enables you to do: set up properties of a particular receiver instance within a lambda that you pass as an argument. For example:

```
val finalBossRoom = EvilLair().apply {
    lairOwner = "Nogartse"
    securityFeatures = listOf("moat", "lasers", "confusing interior layout")
    prepareBattleMusic()
}
```

This allows you to avoid explicitly calling each function on the receiver. Instead, you can call them implicitly within a lambda. The bit of magic that **apply** provides is accomplished by defining a function literal with a receiver.

Looking again at the definition for **apply**, check out how the function parameter called **block** is specified:

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

The **block** function parameter is not only a lambda but also specified as an extension to a generic type **T** with **T.() -> Unit**. This is what allows the lambda that you define to also have access to the receiver instance’s properties and functions implicitly.

Specified as an extension, the lambda’s receiver is also the instance that **apply** is called on – granting access to the receiver instance’s functions and properties within the body lambda.

Using this style, it is possible to write what are known as “domain-specific languages” – an API style that exposes functions and features of a receiver context you configure using lambda expressions that you define to access them. For example, the Exposed framework from JetBrains (github.com/JetBrains/Exposed) makes extensive use of the DSL style for its API to allow you to define SQL queries.

You might add a function to NyetHack that uses this same style, allowing a room to be configured with a pit goblin. (Feel free to add this to your NyetHack project as an experiment.)

```
inline fun MonsterRoom.configurePitGoblin(
    block: MonsterRoom.(Goblin) -> Goblin
): MonsterRoom {
    val goblin = block(Goblin("Pit Goblin", description = "An Evil Pit Goblin"))
    monster = goblin
    return this
}
```

This extension to **Room** accepts a lambda that has **Room** as its receiver. The result is that the properties of **Room** are available within the lambda that you define, so the goblin can be configured using the **Room** receiver’s properties. For example, you could call this extension function in your **Game** class like this:

```
object Game {
    ...

    fun configureCurrentRoom() {
        val monsterRoom = currentRoom as? MonsterRoom ?: return

        monsterRoom.configurePitGoblin { goblin ->
            goblin.healthPoints = when {
                "Haunted" in name -> 60
                "Dungeon" in name -> 45
                "Town Square" in name -> 15
            }
            goblin
        }
    }
}
```

Challenge: Frame Extension

The following is a small program that allows a string of an arbitrary size to be displayed in a beautiful ASCII frame that is suitable for printing and hanging on any wall:

```
fun frame(name: String, padding: Int, formatChar: String = "*"): String {
    val greeting = "$name!"
    val middle = formatChar
        .padEnd(padding)
        .plus(greeting)
        .plus(formatChar.padStart(padding))
    val end = (0 until middle.length).joinToString("") { formatChar }
    return "$end\n$middle\n$end"
}
```

For this challenge, you will apply what you have learned about extensions. Try refactoring the **frame** function as an extension that is available for use with any **String**. An example of calling the new version would look like this:

```
print("Welcome, Madrigal".frame(5))

*****
*      Welcome, Madrigal      *
*****
```

20

Coroutines

Applications perform all kinds of functions, including connecting with outside resources. You may want your app to download data, query a database, or make a request to a web API. These are all useful operations, but they can require a considerable amount of time to complete. You do not want your users to be stuck waiting for an operation to finish before they can continue using your app.

Instead of requiring users to wait while long-running work completes, you should move that work to execute in the background. If you do not, you *block* your program from reacting to any other events, making it look like your program has frozen. [Coroutines](#) let you define work that will run in the background – or, as it is often called, *asynchronously*.

Kotlin 1.3 introduced stable support for coroutines, but coroutines are not new or exclusive to Kotlin. The concept of a coroutine dates back to the 1950s, and they have been implemented in many programming languages. Coroutines are based on the idea of functions being able to *suspend*, meaning that a function can be paused until a long-running operation completes.

Many other programming languages rely exclusively on the concept of a [thread](#) for asynchronous work. Threads are responsible for managing execution of your program. Every thread has its own sequence of instructions that it executes, performing them in the order that they are declared. The primary thread, which manages the work the user interacts with directly, is called the [main thread](#) or *UI thread*.

Traditionally, developers offload long-running work – such as the network request we mentioned earlier – onto a *background thread*. That frees up the main thread to continue executing tasks of its own, like rendering the application's UI. Under this model of asynchronous computing, the main thread might spin up a background thread to start a network request. When the network request finishes, the background thread could send this data back to the main thread for use.

Using threads in this way has a number of disadvantages. Threads are a fairly low-level API, making them difficult to work with. It is also very easy to make mistakes when using threads directly – mistakes that can lead to the application wasting resources or crashing unexpectedly.

Coroutines provide a high-level and safer set of tools to help you build asynchronous code. Under the hood, Kotlin's coroutines use threads to perform work in parallel, but you often do not have to worry about this detail.

When you use a coroutine to perform tasks like making a network request, the code making the request will suspend while the task is performed. When your code is in this suspended state, other parts of your program can still execute, freeing up your main thread and keeping your program responsive. When the network request completes, the suspended code resumes right where it left off.

Behind the scenes, Kotlin saves and restores the function state between suspending function calls. This allows the original function call to be temporarily freed from memory until it is ready to be resumed. Because of these optimizations, coroutines are considerably more resource efficient than native threads.

As you will see in this chapter, a coroutine behaves very similarly to the synchronous code you have already written throughout this book.

Now that her heroic work in Kronstadt is complete, Madrigal is in desperate need of a vacation. To unwind, she has booked an international flight to a tropical island devoid of goblins. To help Madrigal board her flight on time, you will leverage coroutines in a new project to fetch data about her flight and give her instructions about when to board the plane.

By the end of this chapter, your project will interact with two web services: one to get information about Madrigal's flight and the other to get information about her TaernylAir loyalty status, which affects her place in the boarding process. You will combine this information into a single **FlightStatus** object. In [Chapter 21](#), you will generate boarding instructions using flows – streams of data that can be subscribed to. And in [Chapter 22](#), you will speed up your flight tracking using channels, a tool for communication between coroutines.

Blocking Calls

The first web service you will call is hosted at kotlin-book.bignerdranch.com/2e/flight. When data is requested from this service, comma-separated flight data is returned with values for a flight's `flightNumber`, `originAirport`, `destinationAirport`, `status`, and `departureTimeInMinutes` attributes.

Open the URL in your browser. (Be patient. The page has a five-second delay as the system queries its enormous database of flights.) You will see a page with a single line that looks something like this:

```
JC1112,UJH,WUI,On Time,88
```

Reload your web browser several times to see the different responses it provides. The data is randomly generated each time you load the page. In this chapter, you will build a client to consume responses from this API and print them out to the console.

Create a new Kotlin project called `TaernylAir`. (Taernyl has had much success with his tavern and is now branching out into the aviation business.) Remember to use the Application template under the JVM section and to set your Project JDK. Also, be sure to use the Gradle Groovy build system. You will be modifying files related to the build system later in this chapter, and the other build systems will not match these steps.

In your new project, create a new file called `FlightFetcher.kt`. In this file, define two constants called `BASE_URL` and `FLIGHT_ENDPOINT` for the web API endpoints.

Also, create a new function called **`fetchFlight`** that returns a **`String`** representing the data received from the web endpoint. Kotlin includes an extension function to Java's **`URL`** class called **`readText`** that provides simple support for connecting to a basic web API endpoint, buffering the data, and converting that data into a string – everything you need here.

Finally, call **`fetchFlight`** from a new **`main`** function and print out the result.

Listing 20.1 Fetching flight data (`FlightFetcher.kt`)

```
private const val BASE_URL = "http://kotlin-book.bignerdranch.com/2e"
private const val FLIGHT_ENDPOINT = "$BASE_URL/flight"

fun main() {
    val flight = fetchFlight()
    println(flight)
}

fun fetchFlight(): String = URL(FLIGHT_ENDPOINT).readText()
```

Run the **`main`** function in `FlightFetcher.kt`. Depending on the speed of your internet connection, you might notice that this call takes quite some time to return data. When it finishes, add some logging statements so you can see when your request starts and finishes.

Listing 20.2 Measuring request timing (`FlightFetcher.kt`)

```
...
fun main() {
    println("Started")
    val flight = fetchFlight()
    println(flight)
    println("Finished")
}

fun fetchFlight(): String = URL(FLIGHT_ENDPOINT).readText()
```

Run your **`main`** function again and watch the console. Notice the time between when `Started` prints and when `Finished` prints.

Because of the delay on the flight endpoint, your **`fetchFlight`** call takes approximately five seconds to return. A thread is like a pipeline that handles a sequence of work to be performed; while waiting for **`fetchFlight`** to return, your thread is blocked – it cannot be used for any other work until it is freed up.

Five seconds is a long time to wait, but it is not unusual for a response from a web service to take that long, especially over a poor network connection or with a large response body. To make these sorts of long-running calls without wasting your user's time (and patience), you will move this work onto a separate thread. That way, other work can be performed while the long-running task executes.

Enabling Coroutines

Like threads, coroutines are a mechanism for performing asynchronous, potentially long-running work in the background. Unlike threads, coroutines can execute and wait for the completion of other work without blocking the thread they are launched on – thanks to the magic of suspending functions.

Kotlin does not ship with support for coroutines. To use them in your project, you need to add the library where they are defined to your project as a dependency. Dependencies are managed by Gradle, the build system you selected when creating the TaarnylAir project. Gradle projects are primarily configured with files ending in `.gradle`. These files contain information about the Kotlin version you are using, dependencies your project needs, settings that determine how to generate outputs for your program, and much, much more.

For now, you only need to register one dependency with Gradle. Find and open the `build.gradle` file in your top-level project directory and add the Coroutines dependency:

Listing 20.3 Enabling coroutines (`build.gradle`)

```
plugins {  
    id 'org.jetbrains.kotlin.jvm' version '1.5.21'  
}  
  
group = 'com.bignerdranch'  
version = '1.0-SNAPSHOT'  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.1"  
    testImplementation 'org.jetbrains.kotlin:kotlin-test'  
}  
  
test {  
    useJUnit()  
}  
  
compileKotlin {  
    kotlinOptions.jvmTarget = '1.8'  
}  
  
compileTestKotlin {  
    kotlinOptions.jvmTarget = '1.8'  
}
```



Once you add the entry to your `build.gradle` file, click the  Load Gradle Changes button that appears near the top-right corner of the editor to sync the Gradle files.

By the way, the Coroutines library also includes support for the flows and channels you will use in upcoming chapters.

Coroutine Builders

A [coroutine builder](#) is a function that creates a new coroutine. Most coroutine builders also start the coroutine immediately after creating it. Several builders are defined for you in the Coroutines library. The most commonly used coroutine builder is **launch**, a function defined as an extension to a class called **CoroutineScope**. You will see more information on scopes shortly; for now, you will use the subclass **GlobalScope**.

Launch a new coroutine by wrapping your call to **fetchFlight** in a call to the **launch** function defined on **GlobalScope**.

Listing 20.4 Launching a coroutine (FlightFetcher.kt)

```
private const val BASE_URL = "http://kotlin-book.bignerdranch.com/2e"
private const val FLIGHT_ENDPOINT = "$BASE_URL/flight"

fun main() {
    println("Started")
    GlobalScope.launch {
        val flight = fetchFlight()
        println(flight)
    }
    println("Finished")
}

fun fetchFlight(): String = URL(FLIGHT_ENDPOINT).readText()
```

The **launch** function requires one argument: a lambda that specifies the work you want to run without blocking the current thread.

Run your new version of the application. This time, `Started` and `Finished` are printed, but the process finishes before your request returns data. What happened here?

You created a new coroutine by calling the **launch** function. **launch** starts the work that you specify in a new coroutine immediately.

A coroutine’s scope is determined by the **CoroutineScope** in which it is defined. You called **launch** on **GlobalScope**, so this class is your coroutine’s scope and defines how long it will run. **GlobalScope** is an implementation of **CoroutineScope** that is essentially unmanaged: It tosses all its coroutines into a shared [thread pool](#).

You may have noticed a warning that appeared in your code when you used **GlobalScope**. This class can cause problems when used incorrectly. For example, if you launch a coroutine in **GlobalScope** when a user enters part of your app but forget to cancel it when they leave, the associated resources will not be cleaned up and you may have a coroutine that continues churning away in the background unnecessarily.

Because of this, **GlobalScope** is considered a “delicate” API. IntelliJ is trying to guide you to other APIs that are considered safer. To understand this suggestion, you need to explore the concept of a coroutine scope in more depth.

Coroutine Scopes

Every coroutine builder launches its coroutines inside a [coroutine scope](#). A coroutine's scope has control over how the coroutine's code executes. This includes setting up the coroutine, canceling the coroutine, and choosing which thread will be used to run the code. At the moment, you are using **GlobalScope** as the coroutine scope to perform your network requests.

GlobalScope provides an easy way to launch a coroutine, so it is useful as a first example here. But we do not actually recommend it for most applications, since it can introduce problems into your code. In this case, it is the cause of the problem you are seeing: **GlobalScope** does not keep your process alive long enough to wait for the response to return and print out.

Up to this point, we have been talking about asynchronous code as though blocking is always to be avoided, but that is not actually the case. Sometimes, you *want* your code to block so a critical task can complete. In TaernylAir, while you do not want **fetchFlight** to block, you *do* want to block completion of your **main** function until the work in **fetchFlight** completes. This will ensure that **fetchFlight** has a chance to return data.

The **runBlocking** function is a coroutine builder that blocks its thread until execution of its coroutine is complete. You can use **runBlocking** to launch coroutines that must all complete before execution continues. You will see more examples of this technique in [Chapter 21](#) and [Chapter 22](#).

Surround your **launch** function in **main** with a call to **runBlocking**. And now that you are using this builder's coroutine scope, you can remove your use of **GlobalScope**.

Listing 20.5 Using a blocking coroutine builder (FlightFetcher.kt)

```
fun main() {
    println("Started")
    GlobalScope.launch {
        runBlocking {
            println("Started")
            launch {
                val flight = fetchFlight()
                println(flight)
            }
            println("Finished")
        }
        println("Finished")
    }
}

fun fetchFlight(): String = URL(FLIGHT_ENDPOINT).readText()
```

Run TaernylAir. Your output should look something like:

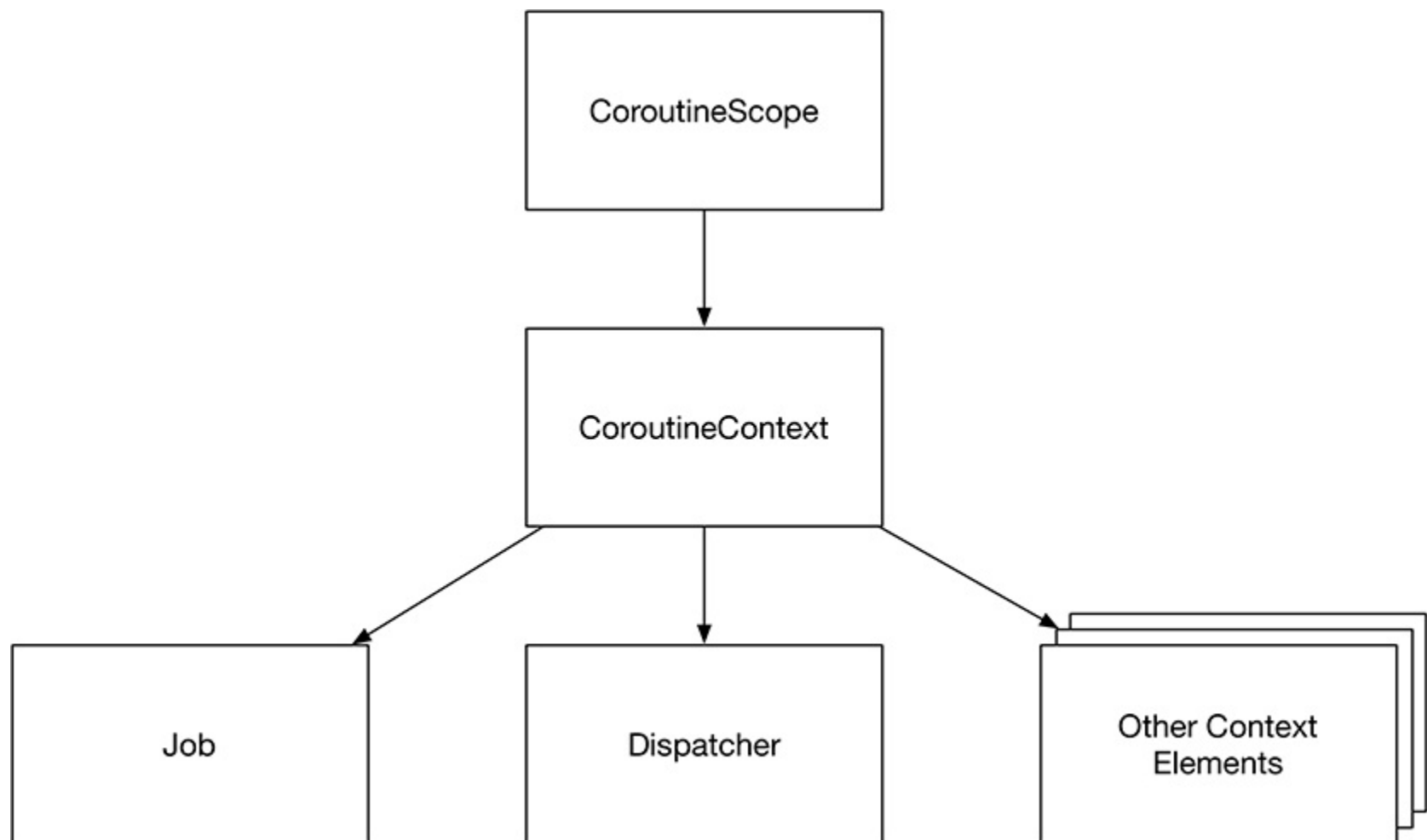
```
Started
Finished    // pause
CE7902,FVY,CLI,On Time,116
```

Started prints first, as its call to **println** comes first and **println** is not a suspending function. The call to **fetchFlight** comes next, but because it was launched in a separate coroutine, the second call to **println** does not wait for it to return. Retrieving the flight data takes considerably more time to execute than printing to the console, so you see **Finished** print before the flight data.

Structured Concurrency

Behind the scenes, each **CoroutineScope** has a **CoroutineContext**. Think of a coroutine context as the set of rules for how a coroutine should execute, and the coroutine scope as the overseer of a coroutine's execution according to those rules. A **CoroutineContext** is further broken down into smaller components that define execution rules. The most commonly used elements that make up a coroutine context are a **Job** and a **CoroutineDispatcher** ([Figure 20.1](#)).

Figure 20.1 Anatomy of a CoroutineScope



A coroutine's **Job** tracks information about the coroutine's state, like whether it is running. It also has controls that allow the coroutine to be canceled early. Each coroutine builder returns the corresponding **Job** when it launches a coroutine. This means that you have access to this execution information and can manually cancel a coroutine before it finishes executing. This is particularly handy if you need to abort a long-running background task that was triggered by a user – for example, if the user leaves a page while it is loading or cancels a file upload.

The **CoroutineDispatcher** is responsible for starting the coroutine's execution. Generally, this involves scheduling or moving the coroutine's work to a desired thread. You have several options for which dispatcher to use, and you can also create your own. The most common built-in dispatchers are in a class called **Dispatchers**:

`Dispatchers.Default`

Recommended for general work and computationally expensive operations. Backed by a thread pool whose size is bounded by the number of the device's processor cores.

`Dispatchers.IO`

Recommended for input/output-related operations. Backed by a thread pool with a high number of threads.

`Dispatchers.Main`

Runs code on the UI or main thread. On the JVM, you need an additional dependency for Android, JavaFX, or Swing to indicate which thread is the main thread. For other platforms, this dispatcher behaves the same as `Dispatchers.Default`.

`Dispatchers.Unconfined`

Specifies that the thread that this work executes on does not matter. The Coroutines library will continue executing the coroutine on whatever thread it is already using.

Each **CoroutineScope** has its own context that will be used to run the coroutine, but you can also modify the

context of a coroutine for a portion of its work. You do this with the **withContext** function. Try it out by making your **fetchFlight** function run with `Dispatchers.IO`. You also need to mark the function with the `suspend` keyword, which we will explain after you make this change.

Listing 20.6 Using a coroutine context (`FlightFetcher.kt`)

```
...
fun main() {
    runBlocking {
        println("Started")
        launch {
            val flight = fetchFlight()
            println(flight)
        }
        println("Finished")
    }
}

suspend fun fetchFlight(): String = withContext(Dispatchers.IO) {
    URL(ENDPOINT).readText()
}
```

Now, your work in **fetchFlight** will be executed on a thread managed by `Dispatchers.IO`. The coroutine you launch in **main** will be executed on the main thread, which blocks until all coroutines have finished.

When you switch the context of a coroutine using **withContext**, the new context is combined with the previous context. This causes the new context to *inherit* from the context of the parent coroutine. In this example, you specified a different dispatcher for the new context, which will override (not combine with) the dispatcher from the parent coroutine's context.

Importantly, child coroutines also inherit the parent's **Job** in addition to getting their own. Unlike the dispatcher, the parent's job does not get overridden – both jobs will be able to control the coroutine's execution. If the parent job is canceled, the jobs of all the children (and the children's children) are also canceled.

This concept is known as [structured concurrency](#), and it frequently occurs when using coroutines in Kotlin. You also see this behavior of inheriting coroutine contexts when you launch a coroutine inside another coroutine, which you are doing when you call **launch** inside **runBlocking**.

Structured concurrency provides a paradigm for organizing your coroutines and is essential as your use of coroutines grows. Although structured concurrency is not exclusive to Kotlin's coroutines, they are one of its best-known implementations.

To call **withContext**, you had to mark **fetchFlight** with the `suspend` modifier. Suspending functions, like **withContext**, can only be called from other suspending functions or inside a coroutine builder. **withContext** will suspend while your networking happens on a different thread. Once the network request finishes, **withContext** will resume execution on your original dispatcher, and **fetchFlight**'s execution will continue where it left off.

There are many reasons for a function to be marked as suspending, and you will be creating your own suspending functions in the next two chapters.



IntelliJ adds a  icon next to where your **main** function calls **fetchFlight**. The same icon also appears on the line with your **withContext** call. This icon indicates that you called a suspending function on that line and serves as a reminder of where your coroutine might pause and resume.

Run TaernylAir and confirm that your output has not changed. The networking will happen on an I/O thread, but because you are waiting for the network request to finish before any other work happens, you will not yet see any benefits in your application's behavior.

Using an HTTP Client

Your TaernylAir project is now capable of loading flight information, but that alone is not enough information get Madrigal onto her plane. There are very strict rules about how passengers are to board. For example, the boarding doors close 15 minutes before the plane departs, and all passengers must be on board if they are to take to the skies.

In addition, Madrigal is part of Taernyl’s loyalty program. Each tier of this program has its own boarding priority, and the tiers are based on how many miles the passenger has flown with the airline.

To show accurate boarding information for a passenger, you will need to query their loyalty tier information from the endpoint at kotlin-book.bignerdranch.com/2e/loyalty.

But before you start adding more API calls, take another look at your **fetchFlight** function. IntelliJ has highlighted your **URL** usage; if you mouse over it you will see a warning that reads Inappropriate blocking method call. The problem is that **URL** and **readText** will block your thread inside a coroutine. By blocking the thread, you will prevent your coroutine from suspending, which ties up resources that the Coroutines library could otherwise use to execute coroutines that are ready to resume.

In practice, this will not affect your code, because you are using `withContext(Dispatchers.IO)`. `Dispatchers.IO` is designed for I/O-based work and has a high number of threads at its disposal, so it is unlikely to cause issues if you block one of the threads in `Dispatchers.IO`’s thread pool. However, you can still heed your IDE’s advice by updating this implementation to use an HTTP client that supports suspending.

There are several libraries that work well with coroutines (including the popular Retrofit library). You will be using Ktor for this project. Ktor is a networking library that is part of Kotlinx – a set of optional, first-party libraries provided by JetBrains that extend on the base functionality in the Kotlin language and standard library. The Coroutines library you are using is also part of Kotlinx.

A unique advantage of Kotlinx libraries is that they are overwhelmingly compatible with Kotlin Multiplatform, making them a great choice for projects that intend to share code across platforms. You will learn more about these code-sharing techniques in [Chapter 24](#).

To enable Ktor in your project, update your `build.gradle` file once again to include the necessary dependencies.

Listing 20.7 Adding Ktor dependency (build.gradle)

```
...
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.1"
    implementation "io.ktor:ktor-client-core:1.6.2"
    implementation "io.ktor:ktor-client-cio:1.6.2"
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}
...
```

The first dependency you declared here will allow you to create an HTTP client with Ktor. The second dependency is for the underlying engine that Ktor will use to perform its network requests. There are several engines you can choose from; CIO (coroutine-based I/O) is a simple library that supports the JVM and does not have any other dependencies.

After updating your `build.gradle` file, you will again need to click the  Load Gradle Changes button before you can use the new dependencies.

With Ktor’s dependencies in place, update your **fetchFlight** function to use an **HttpClient** instead of a **URL**. You can also remove your **withContext** call, because Ktor will automatically move the network request to a background thread and suspend until it completes. (You will need to add import statements for the **HttpClient** and **CIO** classes as well as the **get** function. Make sure you use the correct import directives, otherwise you might see unexpected errors.)

Listing 20.8 Migrating to HttpClient (FlightFetcher.kt)

```
import io.ktor.client.HttpClient
import io.ktor.client.engine.cio.CIO
import io.ktor.client.request.get
import kotlinx.coroutines.*
...
suspend fun fetchFlight(): String = withContext(Dispatchers.IO) {
    URL(FLIGHT_ENDPOINT).readText()
    val client = HttpClient(CIO)
    return client.get<String>(FLIGHT_ENDPOINT)
}
```

Rerun your application and confirm that it still retrieves a flight, as it did before.

With your API calls refactored, you can now implement your second API call, which will fetch Madrigal’s loyalty status. Create another constant for the new endpoint and fetch her loyalty information.

Listing 20.9 Calling your second endpoint (FlightFetcher.kt)

```
private const val BASE_URL = "http://kotlin-book.bignerdranch.com/2e"
private const val FLIGHT_ENDPOINT = "$BASE_URL/flight"
private const val LOYALTY_ENDPOINT = "$BASE_URL/loyalty"

...
suspend fun fetchFlight(): String {
    val client = HttpClient(CIO)
    return val flightResponse = client.get<String>(FLIGHT_ENDPOINT)
    val loyaltyResponse = client.get<String>(LOYALTY_ENDPOINT)

    return "$flightResponse\n$loyaltyResponse"
}
```

Run your application again. You should see output like this:

```
Started
Finished
VA4520,RXF,PBY,On Time,95
Platinum,90781,9218
```

So far, so good. But what is all this data? To make your application more user-friendly and your code more structured, create a new **FlightStatus** class in its own file. This class will parse the results from your endpoints and will keep track of information for both the flight and the passenger, using the app to obtain boarding information. You will take advantage of this model more extensively in [Chapter 21](#).

Listing 20.10 Setting up FlightStatus (FlightStatus.kt)

```
data class FlightStatus(
    val flightNumber: String,
    val passengerName: String,
    val passengerLoyaltyTier: String,
    val originAirport: String,
    val destinationAirport: String,
    val status: String,
    val departureTimeInMinutes: Int
) {

    companion object {
        fun parse(
            flightResponse: String,
            loyaltyResponse: String,
            passengerName: String
        ): FlightStatus {
            val (flightNumber, originAirport, destinationAirport, status,
                departureTimeInMinutes) = flightResponse.split(",")

            val (loyaltyTierName, milesFlown, milesToNextTier) =
                loyaltyResponse.split(",")

            return FlightStatus(
                flightNumber = flightNumber,
                passengerName = passengerName,
                passengerLoyaltyTier = loyaltyTierName,
                originAirport = originAirport,
                destinationAirport = destinationAirport,
                status = status,
                departureTimeInMinutes = departureTimeInMinutes.toInt()
            )
        }
    }
}
```

With your new **FlightStatus** class in place, update your **fetchFlight** function to return the new type and to ask for the passenger’s name.

Listing 20.11 Parsing your flights (FlightFetcher.kt)

```
...
fun main() {
    runBlocking {
        println("Started")
        launch {
            val flight = fetchFlight("Madrigal")
            println(flight)
        }
        println("Finished")
    }
}

suspend fun fetchFlight(passengerName: String): String FlightStatus {
    val client = HttpClient(CIO)
    val flightResponse = client.get<String>(FLIGHT_ENDPOINT)
    val loyaltyResponse = client.get<String>(LOYALTY_ENDPOINT)

    return "$flightResponse\n$loyaltyResponse"
    return FlightStatus.parse(
        passengerName = passengerName,
        flightResponse = flightResponse,
        loyaltyResponse = loyaltyResponse
    )
}
```

Run your application again. You should see that your output has changed and now looks something like this:

```
Started
Finished
FlightStatus(flightNumber=YY8272, passengerName=Madrigal,
    passengerLoyaltyTier=Gold, originAirport=GWX, destinationAirport=LFX,
    status=On Time, departureTimeInMinutes=66)
```

TaernylAir now has all the information it needs to guide passengers onto their flight. But this implementation still leaves something to be desired.

Although you are using suspend functions to avoid blocking your main thread, a coroutine does not change the fact that each of your instructions executes sequentially. A coroutine simply indicates that the work can be paused, wait for some work to finish, and resume at a later time. The statements in the function still execute one at a time and in the order you declared.

In **fetchFlight**, you are making two network requests – but, because of how you have declared them, the request for your loyalty status lookup will not be sent until you finish loading the flight information.

We mentioned earlier that the `flight` endpoint has a delay of five seconds. The `loyalty` endpoint also has a delay, of two seconds. To make this additional delay more visible, add a couple more logs to your **`fetchFlight`** function.

Listing 20.12 Logging responses (`FlightFetcher.kt`)

```
...
suspend fun fetchFlight(passengerName: String): FlightStatus {
    val client = HttpClient(CIO)

    println("Started fetching flight info")
    val flightResponse = client.get<String>(FLIGHT_ENDPOINT).also {
        println("Finished fetching flight info")
    }

    println("Started fetching loyalty info")
    val loyaltyResponse = client.get<String>(LOYALTY_ENDPOINT).also {
        println("Finished fetching loyalty info")
    }

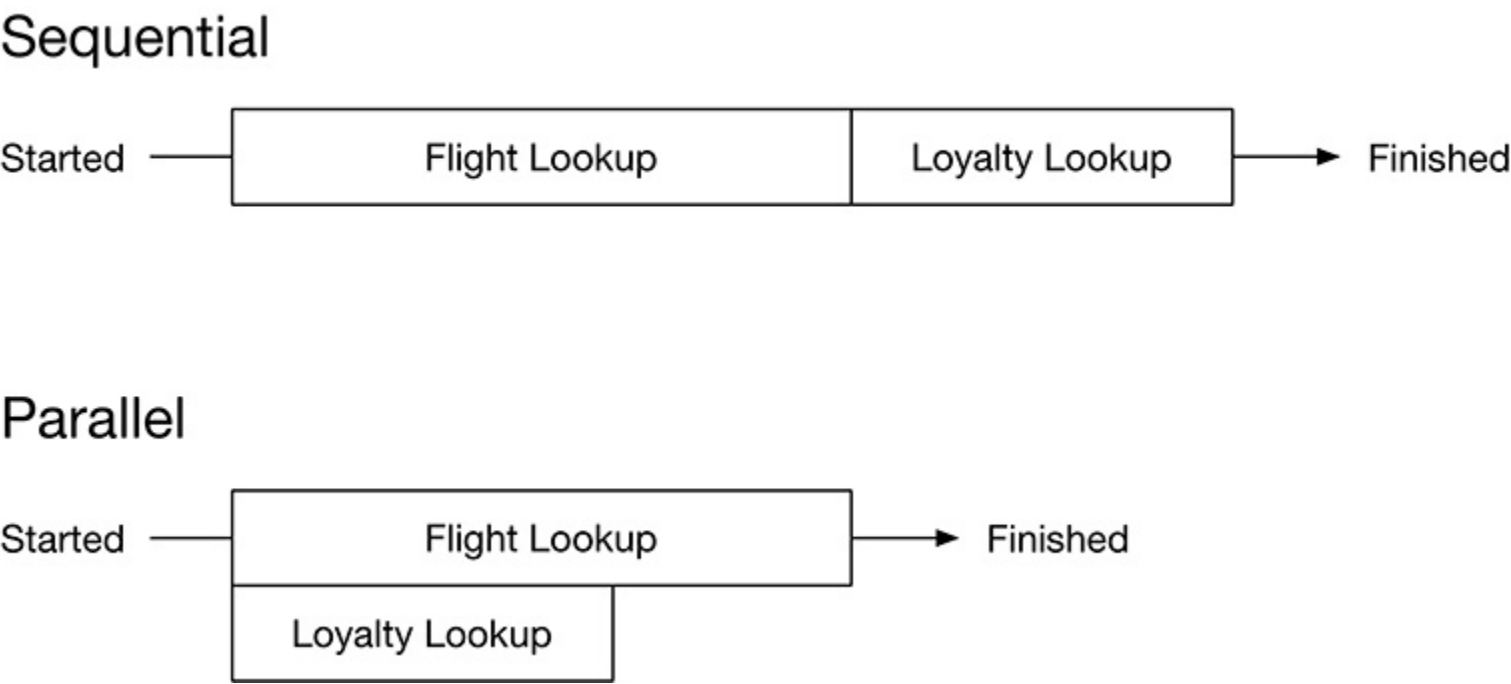
    println("Combining flight data")
    return FlightStatus.parse(...)
}
```

Run your application again. Your output should begin:

```
Started
Finished
Started fetching flight info
Finished fetching flight info
Started fetching loyalty info
Finished fetching loyalty info
Combining flight data
FlightStatus(...)
```

[Figure 20.2](#) shows the timing of your two sequential network requests compared to an alternative approach where the requests to the flight and loyalty endpoints execute at the same time.

Figure 20.2 Sequential vs parallel requests



In the parallel sequence, the loyalty status lookup would start and finish before the flight data returned with a result. If you could structure your network requests in this way, your **`fetchFlight`** function would be faster. And you can. A simple way to accomplish this is with **`async`** and **`await`**.

async and await

async is a coroutine builder that can be used as an alternative to **launch**. Much like **launch**, **async** accepts a lambda expression as an argument, which is where you can call other code that suspends. The big difference between these two functions is their return types: **launch** returns a **Job**, but **async** returns an instance of **Deferred**.

These two classes are similar – in fact, **Deferred** extends from **Job**. But in addition to including information about the status of the coroutine, **Deferred** will also get a hold of the value returned by the coroutine. A **Deferred** value represents a value that might not be ready at this moment, but will be available eventually. The **Deferred** that is returned by **async** will receive whatever value you return from the lambda expression as soon as it is ready.

This is extremely useful, because the work that you pass into **async** can start and execute independently of the rest of the coroutine. You only encounter any remaining delay that the work imposes when you need to access the value.

To access the deferred value, you call **await** on the **Deferred**. This **await** is also a suspend function: When you call it, it will immediately return the result if the work has finished, otherwise it will suspend until the value is ready.

Using **async** and **await**, update **fetchFlight** to perform its network requests in parallel. Also, insert a slight delay before you merge the results to make it easier to see that they are running at the same time.

Listing 20.13 Using async and await (FlightFetcher.kt)

```
...
suspend fun fetchFlight(passengerName: String): FlightStatus = coroutineScope {
    val client = HttpClient(CIO)

    println("Started fetching flight info")
    val flightResponse = async {
        println("Started fetching flight info")
        client.get<String>(FLIGHT_ENDPOINT).also {
            println("Finished fetching flight info")
        }
    }

    println("Started fetching loyalty info")
    val loyaltyResponse = async {
        println("Started fetching loyalty info")
        client.get<String>(LOYALTY_ENDPOINT).also {
            println("Finished fetching loyalty info")
        }
    }

    delay(500)
    println("Combining flight data")
    return FlightStatus.parse(
        passengerName = passengerName,
        flightResponse = flightResponse.await(),
        loyaltyResponse = loyaltyResponse.await()
    )
}
```

Coroutine builders like **async** and **launch** can only be called on a **CoroutineScope** object. Unfortunately, the suspend modifier on the function does not provide direct access to the scope that the function is running in. Instead, you have to get more creative about how you obtain a coroutine scope.

You previously obtained **CoroutineScopes** using **GlobalScope**, **runBlocking**, and **withContext**. None of these techniques is ideal here, as they will either prevent you from taking advantage of structured concurrency or lead to unintended side effects: If you use **GlobalScope** and the coroutine that calls your function gets canceled, any work sent to the **GlobalScope** will not get cleaned up. **withContext** can cause your function to use a different dispatcher than the original calling scope. And **runBlocking** will prevent your function from suspending at all, which is also not desirable.

You want **fetchFlight** to be allowed to suspend, run on the same dispatcher that it was called on, and stop if it is used in a job that gets canceled. **coroutineScope** is one way to satisfy all these needs, which is why you are using it here. When you call **coroutineScope**, a new coroutine scope will be created, but it will inherit the dispatcher of the calling scope and will be added as a child to the calling scope's job. Adding this scope as a child means that it will be stopped if the parent scope is canceled.

This is perfect for your purposes, since it grants access to the **async** coroutine builder while maintaining the dispatchers and respecting the lifecycle-scoping that was specified by the function that called **fetchFlight**.

Run TaernylAir one more time. You should see output that looks like the following, which shows that you are now using the power of coroutines to execute multiple network requests in parallel.

```
Started
Finished
Started fetching flight info
Started fetching loyalty info
Combining flight data
Finished fetching loyalty info
Finished fetching flight info
FlightStatus(flightNumber=GM2813, passengerName=Madrigal,
```

```
passengerLoyaltyTier=Platinum, originAirport=AJA, destinationAirport=IEE,  
status=Canceled, departureTimeInMinutes=52)
```

In this chapter, you wrote asynchronous code that lets your application perform work in parallel using Kotlin's official Coroutines library. In the next chapter, you will add more tools to your coroutines repertoire as you build out TaernylAir's ability to give boarding information to passengers.

For the More Curious: Race Conditions

Whenever you are running code in parallel, you need to be aware of [race conditions](#). A race condition is a scenario in which your program behaves incorrectly when instructions execute at an unexpected time or in an unexpected order.

To see a race condition in action, imagine a booking system that would like to efficiently check in its passengers and count the number of passengers it has processed. If there are 1,000 flights with 75 passengers each, you might decide to use a separate coroutine to process each flight’s 75 passengers. To implement this, you can write code that looks like this. Try it in the REPL.

(When you launch the REPL, you will have the choice of launching it in the context of several modules in your project. Select `TaernylAir.main` – the other modules will not have access to the Coroutines library.)

Listing 20.14 Racing flight check-ins (REPL)

```
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

val passengersPerFlight = 75
val numberOfFlights = 1000
var checkedInPassengers = 0

runBlocking {
    repeat(numberOfFlights) {
        launch(Dispatchers.Default) {
            checkedInPassengers += passengersPerFlight
        }
    }
}

println(checkedInPassengers)
74325
```

Run this code snippet a few times and watch the output. In this scenario, there are 75,000 passengers being checked in. Does this match your output?

Depending on the speed of your computer, you will see a different number each time you run this code, but it will be less than the expected 75,000. (If you consistently see 75,000 being printed, try increasing the number of flights in the system.) Why does this happen?

When you use the `+=` operator in `checkedInPassengers += passengersPerFlight`, your code has to perform three distinct steps: First, it reads the value inside both `checkedInPassengers` and `passengersPerFlight`, then it adds them together, then it writes the resulting value back to the position in memory that stores the `checkedInPassengers` variable.

These steps each take some amount of time, and when you are working with the same variable, it is possible for multiple threads to be in different stages of this operation, resulting in the threads clobbering each other’s work.

To avoid this problem, you have several options:

- run the work synchronously and do not use multiple threads
- restructure your code so your threads do not need to access any shared mutable values
- synchronize your threads to ensure that only one of them can use the `checkedInPassengers` variable at a time
- store `checkedInPassengers` with a thread-safe data structure

Each of these approaches has advantages and disadvantages. For this example, we recommend the last one: using a thread-safe data structure.

There is another Kotlinx library called AtomicFU (pronounced “Atomic Foo,” if you are wondering). Much like Ktor and the Coroutines library itself, AtomicFU is a multiplatform library that can be used on the JVM and beyond.

To set up AtomicFU with your project, you need to register a plugin in your `build.gradle` file. Go ahead and make this change:

Listing 20.15 Adding the AtomicFU plugins (build.gradle)

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlinx:atomicfu-gradle-plugin:0.16.2"
    }
}

plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.5.21'
}

apply plugin: 'kotlinx-atomicfu'
...
```

After loading the Gradle changes, you will have access to the **atomic** function, which can be used to declare atomic references. An atomic reference is a thread-safe data structure that performs complex operations (such as the read-increment-write sequence discussed earlier) *atomically*, meaning that the operation is perceived to execute in a single instruction.

For this class of problem, atomic data structures are a great choice for storing data, because they ensure no data will be lost when multiple threads try modifying the same values. Try them out in the REPL. (Build and restart the REPL to ensure the changes are populated.) The only change from your project code is to the declaration of `checkedInPassengers`, which is now a `val` and uses the **atomic** function to wrap its value in an **AtomicReference**.

Listing 20.16 Using atomics (REPL)

```
import kotlinx.atomicfu.atomic
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

val passengersPerFlight = 75
val numberOfFlights = 1000
val checkedInPassengers = atomic(0)

runBlocking {
    repeat(numberOfFlights) {
        launch(Dispatchers.Default) {
            checkedInPassengers += passengersPerFlight
        }
    }
}

println(checkedInPassengers)
75000
```

Run this code several times. You will see that all 75,000 passengers are consistently checked in, indicating that your race condition is no more.

There is still much to learn about thread safety, though the details are beyond the scope of this book. Atomics worked well in this situation, where multiple threads needed to write to the same field, but they are not always the answer.

Sometimes, you need to block off entire sections of code to ensure only one thread is in a dangerous section of code. This practice is called [mutual exclusion](#), indicating that only one thread may access a region of code. There are several ways to accomplish this, from **Mutexes** and **Semaphores** to the **synchronized** function and platform-specific APIs.

Try to avoid race conditions by limiting the amount of shared mutable state used between threads. We also recommend that you keep an eye out for race conditions in your code and be prepared to do some race condition debugging if you find that a complex multithreaded algorithm is not behaving as expected.

For the More Curious: Server-Side Kotlin

In this chapter, you used Ktor as an HTTP client to consume data from the flight endpoints. Ktor also has another trick up its sleeve: It is also a web server framework for the JVM. In fact, the [kotlin-book.bignerdranch.com/2e](https://book.bignerdranch.com/2e) endpoints you used were hosted with Ktor.

If you are interested in spinning up your own server using Kotlin and Ktor, check out its official website at ktor.io. For reference, here is a simplified version of the **main** function for the server you have been interacting with in this chapter:

```
fun main() = embeddedServer(Netty, port = 8080) {
    routing {
        get("/") {
            call.respondRedirect(
                url = "https://bignerdranch.com/books/"
            )
        }

        route("2e") {
            get("flight") {
                delay(5000)
                call.respond(
                    status = HttpStatusCode.OK,
                    message = FlightSchedule.random().toString()
                )
            }

            get("loyalty") {
                delay(2000)
                call.respond(
                    status = HttpStatusCode.OK,
                    message = LoyaltyStatus.random().toString()
                )
            }
        }
    }
}.start(wait = true)
```

Challenge: No Cancellations

A canceled flight is terrible news for Madrigal’s vacation plans. For this challenge, discard any response with a status of Canceled. Perform new requests until you receive a flight that is either on time or (to Madrigal’s disappointment) delayed.

21

Flows

The structure of your application is defined by how data moves through it. As a developer, you have many decisions to make. Will various components communicate directly? Will sources of data expose a way for other components to subscribe to changes? Should your data consistently move in one direction? Whatever your strategy, intentionality is key.

In this chapter, you will explore Kotlin's built-in stream support and learn how to send data throughout your application. [Flows](#) represent streams of data that can be subscribed to, and they can help you design applications with a unidirectional flow of information that update based on changes to their state.

This often builds on the kind of object-oriented paradigms you used in NyetHack, and it helps you design classes that are more isolated. If part of your program needs to be aware of a change that occurred somewhere else in your code, it can register itself to receive new state values as they come in, rather than asking for updated values if it thinks a value might have changed.

Flows also implement many of the functional programming operations you learned about in [Chapter 11](#), giving you a powerful set of tools to manipulate data.

You will be updating TaernylAir to track multiple flights in succession. For each flight, you will print information about when the plane begins boarding, when the passenger's loyalty group is starting to board, when boarding ends, and when the plane departs.

This simulation is built around events happening in response to other events. By the end of the chapter, your application's state will be made up of two constantly changing pieces of information: which flight is being tracked and the data pertaining to that flight.

You could approach this problem imperatively, using techniques you have seen throughout this book. But an imperative solution would be hard to pull apart to add functionality later. To make your application more modular, you will instead write reactive code using flows.

Setting Up a Flow

Flows are Kotlin’s way to represent asynchronous streams of data. At a high level, flows support two actions: emitting and collecting. When you *collect* a flow, you are registering a listener that will receive all the items that get *emitted* from the flow. A flow can emit any number of items over its lifetime – or it might not emit any items at all.

The specifics of how emission and collection work depend on how the flow is created. Sometimes you will be able to directly manipulate the flow and ask it to emit a specific value. Other times, you will not have any access to the emitting functionality and will only be able to collect from the flow.

In this chapter, you will see examples of two kinds of flows that behave differently. This is only a taste of Kotlin’s flow functionality; there are other ways to create flows that yield different behaviors.

Flows are built into the Coroutines library you already added to your TaernylAir project, so you are ready to put them to use. To build your first flow, create a new file called `FlightWatcher.kt` and add three new functions: a **main** function to organize the flight watching tasks, a second function to observe the status of a flight and print out status updates as the flight gets ready to depart, and a third function to fetch all the flights being tracked.

(When you enter this code, you will need to import both the `kotlinx.coroutines.flow.Flow` class and the `kotlinx.coroutines.flow.flow` function.)

Listing 21.1 Using the flow function (FlightWatcher.kt)

```
fun main() {
    runBlocking {
        println("Getting the latest flight info...")
        val flights = fetchFlights()
        val flightDescriptions = flights.joinToString {
            "${it.passengerName} (${it.flightNumber})"
        }
        println("Found flights for $flightDescriptions")

        flights.forEach {
            watchFlight(it)
        }
    }
}

fun watchFlight(initialFlight: FlightStatus) {
    val currentFlight: Flow<FlightStatus> = flow {
        var flight = initialFlight
        repeat(5) {
            emit(flight)
            delay(1000)
            flight = flight.copy(
                departureTimeInMinutes = flight.departureTimeInMinutes - 1
            )
        }
    }
}

suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
) = passengerNames.map { fetchFlight(it) }
```

The first half of your new **main** function is similar to what you did in [Chapter 20](#); you call the **fetchFlight** function for each passenger, using the **runBlocking** coroutine builder.

The **watchFlight** function is more interesting. Here, you define a flow that will emit new flight data. Each second, the flight’s `departureTimeInMinutes` will be decremented, simulating the flight getting one minute closer to departure.

Think back to the sequence-building function **generateSequence** you saw in [Chapter 11](#). When the sequence it creates with an initial value is acted on by another function, **generateSequence** calls an iterator to determine the next value to produce. Like **generateSequence**, **flow** will lazily emit items in some order to be used by some other component that consumes the items.

Take a look at the call to **emit** in **watchFlight**. This is how you specify the items you want to send to the consumer of the flow. You only have access to this function inside the lambda expression that you provide to **flow**.

emit is a suspend function, like the **delay** function you have used before. But you did not add the **suspend** modifier to **watchFlight**’s function signature, which gives a hint about what is happening here: Behind the scenes, **flow** creates a coroutine scope that will execute the contents of your lambda expression. This is why you can call suspend functions inside the flow’s lambda. The coroutine scope that the flow will use is created when the flow starts being collected and closed when the flow stops emitting items or stops being collected from.

You now have two **main** functions, one in `FlightWatcher.kt` and one in `FlightFetcher.kt`. The compiler does not have a problem with this, and in fact your program can function just fine. But to avoid ambiguity, go ahead and delete the **main** function in `FlightFetcher.kt`, which you no longer need.

Listing 21.2 Removing the old main (FlightFetcher.kt)

```

private const val BASE_URL = "http://kotlin-book.bignerdranch.com/2e"
private const val FLIGHT_ENDPOINT = "$BASE_URL/flight"
private const val LOYALTY_ENDPOINT = "$BASE_URL/loyalty"

fun main() {
    runBlocking {
        println("Started")
        launch {
            val flight = fetchFlight("Madrigal")
            println(flight)
        }
        println("Finished")
    }
}
...

```

If you run `TaernylAir` at this point, none of the code inside your flow’s lambda will be executed. This flow is [cold](#), meaning that it is not emitting data. It will not evaluate the body of your lambda expression until you start collecting from it. (A [hot](#) flow is one that is executing code and emitting values, even when it has no collectors.)

To take advantage of your flow, use the **collect** function. For now, only simulate the first five minutes of boarding after the flight information is fetched. Later in this chapter, you will update this flow’s logic to continue tracking the flight until it departs.

Listing 21.3 Consuming flight data (`FlightWatcher.kt`)

```

...
suspend fun watchFlight(initialFlight: FlightStatus) {
    val passengerName = initialFlight.passengerName
    val currentFlight: Flow<FlightStatus> = flow {
        var flight = initialFlight
        repeat(5) {
            emit(flight)
            delay(1000)
            flight = flight.copy(
                departureTimeInMinutes = flight.departureTimeInMinutes - 1
            )
        }
    }

    currentFlight
        .collect {
            println("$passengerName: $it")
        }
}
...

```

Unlike **flow**, **collect** is a suspend function. To be able to call **collect**, you have to add the **suspend** modifier to **watchFlight**. The **collect** function itself will suspend until the flow stops emitting items. To demonstrate this, add a log after your **collect** call.

Listing 21.4 Waiting for your flow to complete (`FlightWatcher.kt`)

```

...
suspend fun watchFlight(initialFlight: FlightStatus) {
    ...
    currentFlight
        .collect {
            println("$passengerName: $it")
        }

    println("Finished tracking $passengerName's flight")
}

```

Run the **main** function in `FlightWatcher.kt`. You should see output like this:

```

Getting the latest flight info...
...
Finished fetching flight info
Found flights for Madrigal (RD0475), Polarcubis (WG2393)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=110)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=109)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=108)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=107)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=106)
Finished tracking Madrigal's flight

Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=30)
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=29)
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=28)
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=27)
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=26)
Finished tracking Polarcubis's flight

```

So far, so good. The first five minutes of boarding updates are now being printed, but this output leaves something to be desired. `TaernylAir` is not giving the passenger information about whether they can board their plane, and its output is a bit difficult to read. Before you simulate the rest of the boarding process, spend some time updating the output.

To keep track of the logic that determines boarding times, create two new enum classes in your existing `FlightStatus.kt` file:

Listing 21.5 Adding loyalty information and boarding times (`FlightStatus.kt`)

```

data class FlightStatus(
    ...
) {
    ...
}

enum class LoyaltyTier(
    val tierName: String,
    val boardingWindowStart: Int
) {
    Bronze("Bronze", 25),
    Silver("Silver", 25),
    Gold("Gold", 30),
    Platinum("Platinum", 35),
    Titanium("Titanium", 40),
    Diamond("Diamond", 45),
    DiamondPlus("Diamond+", 50),
    DiamondPlusPlus("Diamond++", 60)
}

```

```
enum class BoardingState {
    FlightCanceled,
    BoardingNotStarted,
    WaitingToBoard,
    Boarding,
    BoardingEnded
}
```

LoyaltyTier defines TaernylAir’s loyalty status levels. Each tier has a designated time when its passengers can begin to board. This is stored in `boardingWindowStart` as the number of minutes before the flight departs. So, for example, Gold members can start to board 30 minutes before their flight departs.

To use the first of your new enums, update **FlightStatus** to store its `passengerLoyaltyTier` using a **LoyaltyTier** instead of a **String**.

Listing 21.6 Parsing LoyaltyTiers (FlightStatus.kt)

```
data class FlightStatus(
    val flightNumber: String,
    val passengerName: String,
    val passengerLoyaltyTier: String LoyaltyTier,
    val originAirport: String,
    val destinationAirport: String,
    val status: String,
    val departureTimeInMinutes: Int
) {

    companion object {
        fun parse(
            flightResponse: String,
            loyaltyResponse: String,
            passengerName: String
        ): FlightStatus {
            val (flightNumber, originAirport, destinationAirport, status,
                departureTimeInMinutes) = flightResponse.split(",")

            val (loyaltyTierName, milesFlown, milesToNextTier) =
                loyaltyResponse.split(",")

            return FlightStatus(
                flightNumber = flightNumber,
                passengerName = passengerName,
passengerLoyaltyTier = loyaltyTierName,
                passengerLoyaltyTier = LoyaltyTier.values()
                    .first { it.tierName == loyaltyTierName },
                originAirport = originAirport,
                destinationAirport = destinationAirport,
                status = status,
                departureTimeInMinutes = departureTimeInMinutes.toInt()
            )
        }
    }
}
```

You now have all the information you need to determine a passenger’s boarding status at any given time. Define a new `boardingStatus` property in **FlightStatus** to compute the passenger’s **BoardingState** for the flight. You will need to define four additional properties: `isFlightCanceled`, `hasBoardingStarted`, `isBoardingOver`, and `isEligibleToBoard`. Remember that boarding begins 60 minutes before departure, starting with Diamond++ members, and ends 15 minutes before departure.

Listing 21.7 Computing the BoardingState (FlightStatus.kt)

```
data class FlightStatus(
    ...
) {

    val isFlightCanceled: Boolean
        get() = status.equals("Canceled", ignoreCase = true)

    val hasBoardingStarted: Boolean
        get() = departureTimeInMinutes in 15..60

    val isBoardingOver: Boolean
        get() = departureTimeInMinutes < 15

    val isEligibleToBoard: Boolean
        get() = departureTimeInMinutes in 15..passengerLoyaltyTier.boardingWindowStart

    val boardingStatus: BoardingState
        get() = when {
            isFlightCanceled -> BoardingState.FlightCanceled
            isBoardingOver -> BoardingState.BoardingEnded
            isEligibleToBoard -> BoardingState.Boarding
            hasBoardingStarted -> BoardingState.WaitingToBoard
            else -> BoardingState.BoardingNotStarted
        }
    ...
}
```

With these new properties defined for **FlightStatus**, you can revisit your **watchFlight** function. Replace your use of **repeat** with a while loop to continue providing flight updates until the plane departs. Also, update the formatting of your output to be more helpful to passengers.

Listing 21.8 Improving flight tracking (FlightWatcher.kt)

```
...
suspend fun watchFlight(initialFlight: FlightStatus) {
    val passengerName = initialFlight.passengerName
    val currentFlight: Flow<FlightStatus> = flow {
        var flight = initialFlight
repeat(5){
        while (flight.departureTimeInMinutes >= 0 && !flight.isFlightCanceled) {
            emit(flight)
            delay(1000)
            flight = flight.copy(
                departureTimeInMinutes = flight.departureTimeInMinutes - 1
            )
        }
    }
    currentFlight
        .collect {
            val status = when (it.boardingStatus) {
```



```

        FlightCanceled -> "Your flight was canceled"
        BoardingNotStarted -> "Boarding will start soon"
        WaitingToBoard -> "Other passengers are boarding"
        Boarding -> "You can now board the plane"
        BoardingEnded -> "The boarding doors have closed"
    } + " (Flight departs in ${it.departureTimeInMinutes} minutes)"
    println("$passengerName: $it $status")
}
println("Finished tracking $passengerName's flight")
}
...

```

(To use the values of **BoardingState** without the `BoardingState.` prefix, add `import BoardingState.*` at the top of `FlightWatcher.kt`.)

Run `TaernylAir`. Your output will now look something like this:

```

...
Found flights for Madrigal (0A9084), Polarcubis (YJ8056)
Madrigal: Other passengers are boarding (Flight departs in 34 minutes)
Madrigal: Other passengers are boarding (Flight departs in 33 minutes)
Madrigal: Other passengers are boarding (Flight departs in 32 minutes)
Madrigal: Other passengers are boarding (Flight departs in 31 minutes)
Madrigal: You can now board the plane (Flight departs in 30 minutes)
Madrigal: You can now board the plane (Flight departs in 29 minutes)
Madrigal: You can now board the plane (Flight departs in 28 minutes)
Madrigal: You can now board the plane (Flight departs in 27 minutes)
...
Madrigal: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking Madrigal's flight
...

```

This is a wonderful improvement for passengers. They can now see the state of their flight’s boarding process, and they will continue to receive updates until the flight departs. For your next task, you will be improving `TaernylAir`’s behavior for tracking multiple flights. There may be several flights to track, but only one will be tracked at a time. To inform your users of the overall status, you will be adding logs that indicate how many flights are left to be tracked.

MutableStateFlow

Depending on the state your application needs to manage, you may find that the **flow** builder you have been using is fairly restrictive. Only the lambda expression you provide can emit items into the flow. This worked well to update the departure time of a flight, since the departure time can be updated automatically as time progresses. But other parts of your application will need to allow multiple components to influence their state.

To help users tracking multiple flights, TaernylAir should tell users how many flights are queued to be tracked. The state you need to track for this is the number of flights that are at the gate, waiting for passengers to board. But the **flow** builder is not a good tool for this task, because it is not practical to track this count in one lambda expression.

Instead of writing complex logic to maintain this value in one place, you can store the flight count in a **MutableStateFlow**.

MutableStateFlow is an implementation of **Flow** that is invaluable when it comes to tracking application state. It allows you to create a flow whose value can be reassigned manually after it has been created. Get started by declaring a **MutableStateFlow** for your flight count.

Listing 21.9 Declaring a MutableStateFlow (FlightWatcher.kt)

```
fun main() {
    runBlocking {
        ...
        println("Found flights for $flightDescriptions")
        val flightsAtGate = MutableStateFlow(flights.size)

        flights.forEach {
            watchFlight(it)
        }
    }
    ...
}
```

Notice the differences between your `flightsAtGate` and `currentFlight` flows. When you used **flow** with `currentFlight`, you defined every value the flow would emit over its lifetime. On the other hand, **MutableStateFlow**'s constructor accepts an initial value. If you were to start collecting from this flow right now, you would instantly receive the value of `flights.size`. (With your current code, this value will always be 2.)

Also, a **MutableStateFlow** is a hot flow, while `currentFlight` was a cold flow. A **MutableStateFlow** is always alive – but if you emit to a flow that has no collectors, the values will not be used until **collect** is called.

To make the most of your **MutableStateFlow** (specifically the mutable part), you can update the value kept inside the flow. When you do this, you accomplish two things: The flow will emit the new value to all its active collectors. This means that every component that is registered to observe the value will instantly be notified of the new value. Also, the flow will remember the latest value. If a new consumer starts to collect from the flow, it will immediately receive the most recent value written to the flow.

Update **main** to reduce the `flightsAtGate` count after tracking for a flight has finished.

Listing 21.10 Writing to a MutableStateFlow (FlightWatcher.kt)

```
fun main() {
    runBlocking {
        ...
        val flightsAtGate = MutableStateFlow(flights.size)

        flights.forEach {
            watchFlight(it)
            flightsAtGate.value = flightsAtGate.value - 1
        }
    }
    ...
}
```

Unlike the **emit** function you saw before, the setter for **MutableStateFlow**'s `value` property is not a suspend function. Any part of your code that can access your `flightsAtGate` flow has the power to send values to this flow, even if the code that sends the new value is not running inside a coroutine. This detail becomes important when you define a file-level or class-level property that contains a **MutableStateFlow**.

(You could also use the syntax `flightsAtGate.value--` to decrement this value, but we have shown the full syntax here to demonstrate that you can read and write this value without suspending.)

With great power comes great responsibility, and you may want to restrict who has access to setting the number of flights at the gate. In situations like this, you can take advantage of the **StateFlow** interface. Much like **List** versus **MutableList**, **StateFlow** is the read-only counterpart to **MutableStateFlow**. This is another example of Kotlin's emphasis on immutability.

One common pattern when using **MutableStateFlow** is to expose a read-only version in a public property while keeping the mutable version private. You could accomplish this using two properties, as in the example

below.

```
private val _boardingPass: MutableStateFlow<BoardingPass> =
    mutableStateFlow(BoardingPass())
val boardingPass: StateFlow<BoardingPass>
    get() = _boardingPass

fun refreshBoardingPass() {
    _boardingPass.value = BoardingPass()
}
```

(It is conventional to prefix a private backing property with an underscore when you have both a private and public version of a property that use different types.)

You do not need this for the `flightsAtGate` flow, because it is already scoped inside a function. But for other functionality – like boarding passes, with gate numbers that might change unexpectedly – you could declare the **MutableStateFlow** outside a function and lock off its mutability.

To see the number of flights being tracked, you will need to collect the `flightsAtGate` flow. Add another **collect** call before your passengers begin to take their turns. (You may notice that there is an issue with this example. Do not worry – you will fix it momentarily.)

Listing 21.11 Adding another collect call (FlightWatcher.kt)

```
fun main() {
    runBlocking {
        println("Getting the latest flight info...")
        val flights = fetchFlights()
        val flightDescriptions = flights.joinToString {
            "${it.passengerName} (${it.flightNumber})"
        }
        println("Found flights for $flightDescriptions")

        val flightsAtGate = MutableStateFlow(flights.size)
        flightsAtGate
            .collect { flightCount ->
                println("There are $flightCount flights being tracked")
            }
        println("Finished tracking all flights")

        flights.forEach {
            watchFlight(it)
            flightsAtGate.value = flightsAtGate.value - 1
        }
    }
}
...

```

Run your application again. You will see that it does not progress after printing `There are 2 flights being tracked`. What happened here?

Recall that **collect** is a suspending function – it suspends until the flow has completed. This presents a problem, because your `flightsAtGate` flow has not been told to complete. In fact, the **MutableStateFlow** class is incapable of completing.

What you need is a mechanism for collecting from two flows in parallel. You can accomplish this by moving your calls to **collect** into their own coroutines. Using the **launch** function, update **main** once more to avoid this deadlock issue.

Listing 21.12 Observing Flows in parallel (FlightWatcher.kt)

```
fun main() {
    runBlocking {
        ...
        val flightsAtGate = MutableStateFlow(flights.size)
        launch {
            flightsAtGate
                .collect { flightCount ->
                    println("There are $flightCount flights being tracked")
                }
            println("Finished tracking all flights")
        }

        launch {
            flights.forEach {
                watchFlight(it)
                flightsAtGate.value = flightsAtGate.value - 1
            }
        }
    }
}
...

```

Run `FlightWatcher.kt` again. You should see output including the number of flights being tracked:

```
...
Found flights for Madrigal (ER9618), Polarcubis (M07737)
There are 2 flights being tracked
...
Madrigal: The boarding doors have closed (Flight departs in 3 minutes)
Madrigal: The boarding doors have closed (Flight departs in 2 minutes)
Madrigal: The boarding doors have closed (Flight departs in 1 minutes)
Madrigal: The boarding doors have closed (Flight departs in 0 minutes)
There are 1 flights being tracked
...
Polarcubis: The boarding doors have closed (Flight departs in 3 minutes)
Polarcubis: The boarding doors have closed (Flight departs in 2 minutes)
Polarcubis: The boarding doors have closed (Flight departs in 1 minutes)
Polarcubis: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking all flights

```

Your flight tracking is now working as expected, but `TaernylAir` has one more issue to address. After printing `Finished tracking all flights`, your program does not terminate. For your next task, you will update your code so `TaernylAir` finishes execution after the last flight has departed.

Flow Termination

You have seen examples of flows that automatically complete and flows that are incapable of completing. There are also several tools you have at your disposal that can influence or change the way that a flow completes.

If a coroutine is canceled before the flow completes, the collector will stop receiving emissions and the flow will become dormant, if it has no other collectors. To modify this behavior, you can use an [operator](#). Operators change how a flow emits items. (Although they have the same name, they are different from the operators like + and - you have seen throughout this book.)

Depending on which operator you use, you can omit, add, or modify items in a flow or – for your current needs – change how the flow terminates. Here are a few operators that affect a flow’s termination, some of which you may recognize from Kotlin’s collections and sequence APIs. These operators are defined in a file called `Limit.kt` in the Coroutines library.

- take** Accepts an **Int** parameter and collects up to that number of emissions, then terminates.
- takeWhile** Collects emissions until an item is emitted that does not match the provided **predicate**, then terminates.
- drop** Accepts an **Int** parameter and ignores up to that number of emissions from the flow, then forwards all other values to be collected.
- dropWhile** Ignores emissions until an item is emitted that does not match the provided **predicate**, then forwards all other values to be collected.

These functions are also available on the collection and sequence types you learned about in [Chapter 11](#). In fact, many flow operators also exist as functional programming operations on Kotlin’s collection and sequence types. The most commonly used operators are shown in **Flow**’s documentation, which you can find at kotlinlang.org/docs/flow.html.

To let `TaernylAir` terminate, you need to update your usage of `flightsAtGate` to stop collecting after the count reaches zero. You can accomplish this using the **takeWhile** operator.

Listing 21.13 Canceling an endless flow (`FlightWatcher.kt`)

```
fun main() {
    runBlocking {
        ...
        val flightsAtGate = MutableStateFlow(flights.size)
        launch {
            flightsAtGate
                .takeWhile { it > 0 }
                .collect { flightCount ->
                    println("There are $flightCount flights being tracked")
                }
            println("Finished tracking all flights")
        }
        ...
    }
}
```

Run `FlightWatcher.kt` and let it progress through its output. You should see `Process finished with exit code 0` at the end, indicating that your flows are terminating correctly.

Flow also has a function called **onCompletion** that lets you specify an action you want to take after the flow completes. Clean up your flows by moving the two **println** calls into an **onCompletion** call.

Listing 21.14 Using `onCompletion` (`FlightWatcher.kt`)

```
fun main() {
    runBlocking {
        ...
        val flightsAtGate = MutableStateFlow(flights.size)
        launch {
            flightsAtGate
                .takeWhile { it > 0 }
                .onCompletion {
                    println("Finished tracking all flights")
                }
                .collect { flightCount ->
                    println("There are $flightCount flights being tracked")
                }
            println("Finished tracking all flights")
        }
        ...
    }
}

suspend fun watchFlight(initialFlight: FlightStatus) {
    ...
    currentFlight
        .onCompletion {
            println("Finished tracking $passengerName's flight")
        }
        .collect {
            val status = when (it.boardingStatus) {
                FlightCanceled -> "Your flight was canceled"
                BoardingNotStarted -> "Boarding will start soon"
            }
        }
}
```

```
        WaitingToBoard -> "Other passengers are boarding"
        Boarding -> "You can now board the plane"
        BoardingEnded -> "The boarding doors have closed"
    } + " (Flight departs in ${it.departureTimeInMinutes} minutes)"
    println("$passengerName: $status")
}
println("Finished tracking $passengerName's flight")
}
```

Because the **collect** function already suspends until it completes, this change will not affect the behavior of your code. (You can run it again to confirm this, if you want.) It is simply a nice way to clean up this logic and to associate side-effects with the flow itself.

Flow Transformations

In [Chapter 11](#), you learned about a function called **map** that transforms each item in a collection based on the provided transformer lambda. This function is also an operator you can use with **Flow**. Try it out with your `currentFlight` flow to move some of your logic outside the **collect** lambda.

Listing 21.15 Mapping values in a flow (FlightWatcher.kt)

```
...
suspend fun watchFlight(initialFlight: FlightStatus) {
    ...
    currentFlight
        .map { flight ->
            when (flight.boardingStatus) {
                FlightCanceled -> "Your flight was canceled"
                BoardingNotStarted -> "Boarding will start soon"
                WaitingToBoard -> "Other passengers are boarding"
                Boarding -> "You can now board the plane"
                BoardingEnded -> "The boarding doors have closed"
            } + " (Flight departs in ${flight.departureTimeInMinutes} minutes)"
        }
        .onCompletion {
            println("Finished tracking $passengerName's flight")
        }
        .collect { status ->
            val status = when (it.boardingStatus) {
            FlightCanceled -> "Your flight was canceled"
            BoardingNotStarted -> "Boarding will start soon"
            WaitingToBoard -> "Other passengers are boarding"
            Boarding -> "You can now board the plane"
            BoardingEnded -> "The boarding doors have closed"
            } + " (Flight departs in ${it.departureTimeInMinutes} minutes)"
            println("$passengerName: $status")
        }
    }
    ...
}
```

The **map** function here behaves the same as when you used it in [Chapter 11](#). Every value that gets emitted by this flow will be mapped before it is collected. The collector will only receive the mapped values from the flow.

Many of the transform functions you have seen in Kotlin’s standard library (particularly on the collection types) also make an appearance with **Flow**. For example, **flatMap**, **filter**, and **zip** are all available. (By the way, many developers use the terms “operator” and “transform function” interchangeably, because these functions are so similar. Behind the scenes, though, **Flow** implements its own operators that align very closely to the collection transform functions.)

If you need to define a custom transformation for your flow, you can use **transform**. For example, you could mimic the behavior of the **map** operator in code like this:

```
suspend fun observeTemperature(
    val temperatureInCelsius: Flow<Int>
) {
    temperatureInCelsius.convertToKelvin()
        .collect { temperatureInKelvin ->
            println("The current temperature is $temperatureInKelvin")
        }
}

fun Flow<Int>.convertToKelvin(): Flow<Double> =
    transform<Int, Double> { temperatureInCelsius ->
        emit(temperatureInCelsius + 273.15)
    }
```

This code calls **emit** inside **transform**’s lambda argument. Much like the way you use the **flow** builder, this is how you emit items to whatever component collects the transformed value. If you wanted to omit temperatures that are below absolute zero, you could update the transformation:

```
fun Flow<Int>.convertToKelvin(): Flow<Double> =
    transform<Int, Double> { temperatureInCelsius ->
        val temperatureInKelvin = temperatureInCelsius + 273.15
        if (temperatureInKelvin >= 0) {
            emit(temperatureInKelvin)
        }
    }
```

This one transformation is equivalent to writing a map and then a filter. It converts the temperature to Kelvin, but only emits temperatures that are not negative.

When working with flows, you control how your reactive stream behaves. Almost all the transformations you will want to perform are either built into the Coroutines library (where **Flow** is also published) or can be implemented with a combination of operators. We encourage you to dig into the standard library to discover the other operators that are available out of the box. But if none of them suffice, you can always create your own.

Error Handling in Flows

You have now seen how flows can complete gracefully, but they can also complete unsuccessfully by throwing an exception. When this happens, the exception will be passed until it reaches your **collect** function. By default, **collect** will re-throw any uncaught exceptions in the flow. If you do not have error handling in place, this can cause your application to crash.

To see this behavior in action, add a precondition check to ensure that none of the passengers being tracked has been banned from the airport. Update your **main** function to see what happens when trying to track a flight for Nogartse, who is banned from the airport after a prior incident involving world-eating.

Listing 21.16 Throwing an exception in a flow (FlightWatcher.kt)

```
val bannedPassengers = setOf("Nogartse")

fun main() {
    runBlocking {
        println("Getting the latest flight info...")
        val flights = fetchFlights(listOf("Nogartse"))
        val flightDescriptions = flights.joinToString {
            "${it.passengerName} (${it.flightNumber})"
        }
        println("Found flights for $flightDescriptions")
        ...
    }
}

suspend fun watchFlight(initialFlight: FlightStatus) {
    val passengerName = initialFlight.passengerName

    val currentFlight: Flow<FlightStatus> = flow {
        require(passengerName !in bannedPassengers) {
            "Cannot track $passengerName's flight. They are banned from the airport."
        }

        var flight = initialFlight
        while (flight.departureTimeInMinutes >= 0 && !flight.isFlightCanceled) {
            emit(flight)
            delay(1000)
            flight = flight.copy(
                departureTimeInMinutes = flight.departureTimeInMinutes - 1
            )
        }
    }
    ...
}
```

Run your application again. You will see that it crashes and outputs a stack trace like the following:

```
...
Exception in thread "main" java.lang.IllegalArgumentException: Cannot track
Nogartse's flight. They are banned from the airport.
    at FlightWatcherKt$watchFlight$currentFlight$1.invokeSuspend
        (FlightWatcher.kt:41)
    at FlightWatcherKt$watchFlight$currentFlight$1.invoke(FlightWatcher.kt)
    at FlightWatcherKt$watchFlight$currentFlight$1.invoke(FlightWatcher.kt)
    at kotlinx.coroutines.flow.SafeFlow.collectSafely(Builders.kt:61)
    at kotlinx.coroutines.flow.AbstractFlow.collect(Flow.kt:212)
    ...
```

If you would like to handle exceptions, you have a few tools at your disposal. One option is to wrap the **collect** function in a try/catch block, which would look like this:

```
try {
    currentFlight.collect { println("Got flight data: $it") }
} catch (e: IllegalArgumentException) {
    // Error recovery logic
}
```

Flow also has a **catch** operator, which can intercept errors in the stream and potentially recover from them. The **catch** operator will intercept every type of exception that appears in the stream. It accepts a lambda argument that defines what the flow should do after catching an exception. This lambda behaves the same as the one you saw with **transform** and has access to the same **emit** functions.

The flow will complete after this lambda expression returns and the **catch** block stops emitting items. If you choose to incorporate the **catch** operator, your code would look like this:

```
currentFlight
    .catch { throwable ->
        throwable.printStackTrace()
        emit(/* Fallback value */)
    }
    .collect { println("Got flight data: $it") }
```

For the exception that you are working with, terminating with an error for banned passengers is likely an OK behavior, since you do not want to support such users. Update your **main** function to track flights for Madrigal again.

Listing 21.17 Preventing your crash (FlightWatcher.kt)

```
...
fun main() {
    runBlocking {
        println("Getting the latest flight info...")
        val flights = fetchFlights(listOf("Nogartse"))
        val flightDescriptions = flights.joinToString {
            "${it.passengerName} (${it.flightNumber})"
        }
        println("Found flights for $flightDescriptions")
        ...
    }
}
...
```

Run `FlightWatcher.kt` to make sure it no longer crashes.

Whether you are adopting functional programming or would like a solution to more consistently communicate between classes in your object-oriented application, you may find that **Flows** serve an important role for you. The wealth of operators they provide augments what is provided in the Kotlin standard library, and we encourage you to dig deeper to see what they can do for you.

For the More Curious: SharedFlow

In this chapter, you saw two ways of creating a flow: using the **flow** builder and the **MutableStateFlow** type. These two approaches cover the vast majority of use cases, but for special cases there is another class called **MutableSharedFlow** (and its corresponding read-only type, **SharedFlow**).

MutableSharedFlow is a more generic version of **MutableStateFlow**. In fact, **MutableStateFlow** extends from **MutableSharedFlow**. So, what is it good for?

Suppose you want to collect from a flow with two collectors. The first collector will be added right after you create the flow, but the second collector will not be added until some time after. If you used the **flow** function to build the flow, you would find that both collectors received different values and were not kept in sync.

Try this out in the REPL:

Listing 21.18 Attempting to share with the flow builder (REPL)

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

runBlocking {
    val numbersFlow = flow {
        (1..5).forEach {
            delay(1000)
            emit(it)
        }
    }

    launch {
        numbersFlow.collect { println("Collector 1: Got $it"\n) }
    }

    launch {
        delay(2200)
        numbersFlow.collect { println("Collector 2: Got $it"\n) }
    }
}
Collector 1: Got 1
Collector 1: Got 2
Collector 1: Got 3
Collector 2: Got 1
Collector 1: Got 4
Collector 2: Got 2
Collector 1: Got 5
Collector 2: Got 3
Collector 2: Got 4
Collector 2: Got 5
```

Each collector of a flow built using the **flow** builder has its own state, and new collectors behave as if the flow had never been collected from. If you want to send new values – but no past values – to a new collector, you will find that your current tool belt is not quite prepared for this task. As you have just seen, if you use the the **flow** builder your second collector will start over from the beginning and will not be kept in sync with the previous collector. And if you try to use **MutableStateFlow**, you will see that although collections *are* kept in sync, your second collector will immediately get the last emitted value when it starts collecting – which you do not want in this case, because you do not want any past values to be collected.

MutableSharedFlow, however, is a perfect fit for the job.

As the name indicates, **MutableSharedFlow** shares emissions across all its collectors. Much like **MutableStateFlow**, it is hot and never terminates. But collectors of a **MutableSharedFlow** get only emissions that are sent after they begin collecting.

Take it for a spin in the REPL in place of the **flow** builder.

Listing 21.19 Using MutableSharedFlow (REPL)

```
runBlocking {
    val numbersFlow = MutableSharedFlow<Int>()

    launch {
        numbersFlow.collect { println("Collector 1: Got $it"\n) }
    }

    launch {
        delay(2200)
        numbersFlow.collect { println("Collector 2: Got $it"\n) }
    }

    (1..5).forEach {
        delay(1000)
        numbersFlow.emit(it)
    }
}
Collector 1: Got 1
Collector 1: Got 2
Collector 1: Got 3
Collector 2: Got 3
Collector 1: Got 4
Collector 2: Got 4
Collector 1: Got 5
Collector 2: Got 5
```

Now, the collectors stay in sync and receive the same values.

Notice that you start to collect before emitting any items into the flow. **SharedFlows**, as we said, are hot, much like **MutableStateFlow**. But unlike **MutableStateFlow**, **SharedFlow** does not buffer or re-emit past

values to new collectors. If you had placed the **emit** call before the calls to **collect**, neither of the collectors would have received the first emission.

You can customize this behavior, if you want. **MutableSharedFlow**'s constructor accepts three parameters. The first parameter, **replay**, specifies the number of values that should be remembered and sent to collectors that are added in the future (the default is zero).

The second parameter, **extraBufferCapacity**, specifies additional values that can be buffered in the event that the flow does not currently have collectors (also zero by default, causing values to be discarded when there are no collectors).

The third parameter, **onBufferOverflow**, is only used when **extraBufferCapacity** is at least 1. This parameter controls the behavior when calling **emit** on the flow when its buffer is full. Its default behavior is to suspend until the buffer is empty.

You can mimic the behavior of a **MutableStateFlow** using a **MutableSharedFlow** by setting **replay** to 1 and leaving the other arguments at their defaults.

In general, **MutableStateFlow** and the **flow** builder are the most common ways to create flows. If you are building a flow for a single collector and can define the lifetime of its emissions in one place, go with the **flow** builder. **MutableStateFlow** works well for most other cases – but if you find yourself limited by the behavior of **MutableStateFlow**, look no further than **MutableSharedFlow**.

22

Channels

In the previous chapter, you used **Flow** to build streams that held your flight data as it changed over time. Flows allow you to store application state in an observable property and react to it as its value changes. But sometimes, it is not just two *components* that need to communicate with one another, but two *coroutines*.

If you want to send messages between two coroutines, **Flow** has some limits. Flows offer no way for a sender to know for sure that an emitted value was collected. And if there are multiple collectors, they can all get the same value. In some cases, you want to ensure that a value is received by exactly one coroutine, which you cannot easily accomplish with a flow.

To solve the problem of cross-coroutine communication, look no further than [channels](#). A channel is a communication pathway that has senders and receivers. When a sender outputs a message into the channel, it must wait until that message makes its way to a receiver. Similarly, when a receiver wants to obtain a message from the channel, it must wait for a sender to put one into the channel.

A receiver can only receive one message at a time. If there are multiple receivers that are waiting for a message, only one of them will receive the message – the rest will keep waiting. (If you are familiar with the concept of a blocking queue, you might notice that channels have many similarities.)

There are many ways you can take advantage of this behavior. In this chapter, you will use channels to speed up the loading of the group of flights being tracked.

Dividing Work with Channels

To speed up flight fetching, you will again increase the number of network requests that your application makes in parallel. Previously, you used **async** and **await** for this task. This approach served you well before, since there were a small number of parallel requests that needed to be combined in a straightforward way.

But there are downsides to using **async** and **await** to fetch flights. At a high level, the problems boil down to how you manage your requests: **async** and **await** execute the network requests as soon as possible. If you had many flights to track, this would cause you to send many network requests at the same time, which might overwhelm your network or server.

You will limit your application to fetching two flights at a time. Because each flight needs both the flight information and loyalty status – and because you are already performing those two network requests at the same time using **async** and **await** – your application will now make up to four network requests in parallel (up from the previous two). This will increase the speed of getting flight information for your user while still ensuring that your application does not overwhelm Taernyl’s servers and the user’s network connection.

Setting up this parallelism can be a complex task. How do you know how many flights are being fetched right now? When will you be able to start working on the next flight? What tools will let you efficiently wait for the right conditions to start the next request?

To coordinate the flight loading, you will use three coroutines. The first will be responsible for producing and delegating work. The second and third coroutine will be workers – they will wait for the producer to request that a flight be fetched.

When the producer makes a request, one of the two worker coroutines will consume the request and begin fetching the flight. The other worker will wait for the next request. This sequence of operations is shown in [Figure 22.1](#), contrasted with your current implementation using a single coroutine in a `foreach` loop to fetch each flight in sequence.

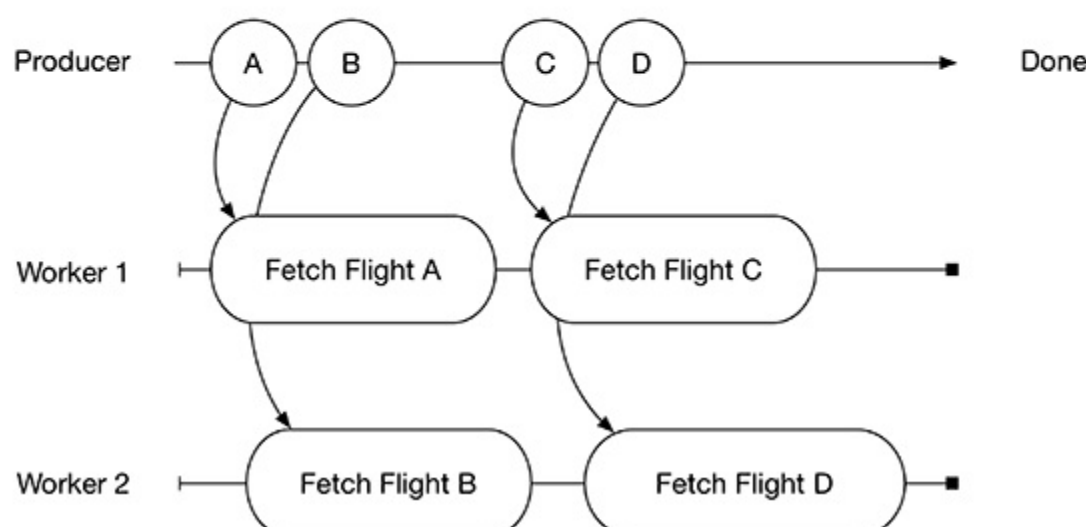
Although this sounds complex, you will see how channels give you the power to concisely implement the interaction among these three coroutines. The result is that the group of flights being tracked will be fetched in roughly half the time, because two workers will fetch them in parallel.

Figure 22.1 One vs multiple workers

Current scenario: `foreach` loop



At the end of this chapter: multiple workers



Sending to a Channel

You will begin with the producer. Your producer will be a separate coroutine that puts requests to fetch flights into a channel. Update your **fetchFlights** function in `FlightWatcher.kt` with a channel for the work requests, and return a temporary result as you continue to build this implementation.

Listing 22.1 Sending requests through a channel (`FlightWatcher.kt`)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
) = passengerNames.map { fetchFlight(it) }
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
    }

    emptyList()
}
```

Be sure to import the `kotlinx.coroutines` **Channel** class.

The `passengerNamesChannel` you created will hold your work requests. The requests themselves will be the passenger names for the flights being fetched, which is the only input to the **fetchFlights** function you created in [Chapter 20](#).

After the channel is created, you launch the producer coroutine. This coroutine will put each of the flight names into the channel.

Take a closer look at the **send** call, which IntelliJ marks as a suspend function. **send** suspends until the value being sent has been received by another coroutine. If you run your code right now, "Madrigal" would be the only value that gets sent to the channel. Nothing is receiving values out of this channel, so the **send** call will suspend forever. Your next task is to build a worker that receives values from this channel.

Receiving from a Channel

With your producer in place, you can create your first worker. Add a function to define the worker’s behavior, then launch it in its own coroutine.

Listing 22.2 Receiving from a channel (FlightWatcher.kt)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
    }

    launch {
        fetchFlightStatuses(passengerNamesChannel)
    }
}

emptyList()

suspend fun fetchFlightStatuses(
    fetchChannel: Channel<String>
) {
    val passengerName = fetchChannel.receive()
    val flight = fetchFlight(passengerName)
    println("Fetched flight: $flight")
}
```

receive is another suspend function. Much like **send**, it will suspend until a value has been sent to the channel. If two coroutines are attempting to receive from the same channel at the same time, only one of them will get the value.

There are no guarantees about which coroutine will receive the value in this situation, but this will not matter for the sake of this chapter. All you care about is that the work happens once, and channels handle that detail for you automatically.

Run TaernylAir. Your output will look something like this:

```
Getting the latest flight info...
Started fetching flight info
Started fetching loyalty info
Combining flight data
Finished fetching loyalty info
Finished fetching flight stats
Fetched flight: FlightStatus(flightNumber=MK1737, passengerName=Madrigal, ...)
```

Only one flight was fetched, but there were two passengers you wanted to track. Plus, your program will never start the simulation or exit on its own. It will remain in this state forever. Press the  stop button to terminate it. What happened here?

The **receive** function only returns one value from the channel. Your worker needs to continue processing requests while there is still outstanding work, so this one **receive** call will not cut it. One way to handle this is to use a for loop to continue receiving items from the channel until there are none left, suspending emissions in between.

Go ahead and add a for loop in **fetchFlightStatuses**:

Listing 22.3 Receiving all values in a channel (FlightWatcher.kt)

```
...
suspend fun fetchFlightStatuses(
    fetchChannel: Channel<String>
) {
    val passengerName = fetchChannel.receive()
    for (passengerName in fetchChannel) {
        val flight = fetchFlight(passengerName)
        println("Fetched flight: $flight")
    }
}
```

Run TaernylAir again. Your output will now look like this.

```
Getting the latest flight info...
Started fetching flight info
Started fetching loyalty info
Combining flight data
Finished fetching loyalty info
Finished fetching flight stats
Fetched flight: FlightStatus(flightNumber=GZ2871, passengerName=Madrigal, ..)
Started fetching flight info
Started fetching loyalty info
Combining flight data
Finished fetching loyalty info
Finished fetching flight stats
Fetched flight: FlightStatus(flightNumber=EH0675, passengerName=Polarcubis, ...)
```

Now, flights are being fetched for both Madrigal and Polarcubis – but the simulation still does not start. You also will not see faster results yet, since you only have one worker and therefore no additional parallelism.

Before you can add a second worker, there are a few loose ends you need to tie up. The biggest issue is that your worker does not do anything after fetching a flight. The result is merely printed and then discarded. The worker needs someplace to put the results when they are ready.

To accomplish this, you will use another channel, which will be responsible for sending fetched flights to the right place. This second channel will contain the **FlightStatus** objects when they are ready.

Update your **fetchFlights** and **fetchFlightStatuses** functions to create a second channel, send resolved flights to the new channel, and implement the return value of **fetchFlights**.

Listing 22.4 Sending results from a worker (FlightWatcher.kt)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    val fetchedFlightsChannel = Channel<FlightStatus>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
    }

    launch {
        fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
    }

    emptyList()
    fetchedFlightsChannel.toList()
}

suspend fun fetchFlightStatuses(
    fetchChannel: Channel<String>,
    resultChannel: Channel<FlightStatus>
) {
    for (passengerName in fetchChannel) {
        val flight = fetchFlight(passengerName)
        println("Fetched flight: $flight")
        resultChannel.send(flight)
    }
}
```

To improve the readability of **fetchFlightStatuses**, you can also take advantage of the `SendChannel` and `ReceiveChannel` types. As the names imply, `SendChannel` is an interface that is only capable of sending values into a channel, and `ReceiveChannel` is an interface that is only capable of receiving items from a channel.

Channel implements both of these interfaces. You can cast a **Channel** to one of these types to narrow the scope of what it can be used for. This can be very helpful to make sure you are not accidentally sending values to or reading values from the wrong channel. Spend another moment updating the **fetchFlightStatuses** function.

Listing 22.5 Narrowing the channels' scope (FlightWatcher.kt)

```
...
suspend fun fetchFlightStatuses(
    fetchChannel: ReceiveChannel<String>,
    resultChannel: SendChannel<FlightStatus>
) {
    for (passengerName in fetchChannel) {
        val flight = fetchFlight(passengerName)
        println("Fetched flight: $flight")
        resultChannel.send(flight)
    }
}
```

These new types help ensure that your inputs and outputs are moving in the right direction. Now, if your worker attempted to dispatch new work out of turn by calling **send** on the `fetchChannel`, you would see a compiler error, because you cannot access the **send** function.

Run `TaernylAir` again. While things have improved behind the scenes, you will see the same behavior as before: Two flights will be printed, but your simulation will not begin and your program will wait indefinitely. To understand what went wrong, we need to discuss how to close a channel.

Closing a Channel

You saw in [Chapter 21](#) that flows can complete, meaning they emit no more items. And flow termination can happen in a few different ways; in the case of the **flow** builder that accepts a lambda expression, the flow implicitly completes when the lambda expression returns.

Similarly, channels can close, so that no additional values can be sent or received through the channel. But the channels you have created so far have no way to implicitly close themselves. You need to explicitly call **close** on the channels when you are finished emitting items.

You will need two calls to **close**. The first **close** call will be on `passengerNamesChannel`. Closing this channel will allow your workers to complete. If this channel does not close, your for loop will suspend indefinitely while it waits for another value to be sent through the channel.

You will also need to close the `fetchFlightsChannel`. Closing this channel indicates to the **toList()** function that all items have been received. This allows the list to be finalized and the **fetchFlights** function to return.

Add two calls to **close** in **fetchFlights**.

Listing 22.6 Closing channels (`FlightWatcher.kt`)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    val fetchedFlightsChannel = Channel<FlightStatus>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
        passengerNamesChannel.close()
    }

    launch {
        fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
        fetchedFlightsChannel.close()
    }

    fetchedFlightsChannel.toList()
}
...
```

Run `TaernylAir` once again. Both flights will be fetched, and the simulation will start. Your program will also stop itself after the simulation completes, instead of pausing forever. The output will match what you saw at the end of the previous chapter.

Remember that the goal is to fetch two flights in parallel. Because you only have one worker, you will not see any speed benefit at this point. Although it feels like you have not made progress, you now have all the necessary scaffolding to add your second worker.

Joining Jobs

With your producer and first worker in place, you are ready for an additional worker. Add a parameter to control the number of workers that you are allowed to use, then spin up additional worker coroutines. Also add two travelers to the list of passengers being tracked so there are more flights for your simulation.

Listing 22.7 Launching multiple workers (FlightWatcher.kt)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis",
        "Estragon", "Taernyl"),
    numberOfWorkers: Int = 2
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    val fetchedFlightsChannel = Channel<FlightStatus>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
        passengerNamesChannel.close()
    }

    repeat(numberOfWorkers) {
        launch {
            fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
            fetchedFlightsChannel.close()
        }
    }

    fetchedFlightsChannel.toList()
}
...
```

Run TaernylAir after making these changes. It will begin to fetch flights, but will crash with the following error after fetching the fourth flight:

```
ClosedSendChannelException: Channel was closed
```

The problem here is that `fetchedFlightsChannel` is closed too early. Whichever worker finishes first will also close the channel, making it impossible for the other workers to send values.

To fix this, you need to wait for all workers to finish before closing the `fetchedFlightsChannel`. Because these network requests take an arbitrary amount of time – and because there are a variable number of requests – there are no guarantees about which worker will finish last. You will need to keep track of which workers are still alive and wait for them all to complete.

Luckily, you have already seen a tool that can take care of this for you. Recall that coroutine builders like **launch** return a **Job**. Jobs contain information about the state of the coroutine and allow you to do things like cancel the coroutine early.

Jobs also have a suspend function called **join**. When called, it will suspend until the job finishes. If the job has already completed, **join** will not suspend, and the rest of your code will continue to execute.

Kotlin also includes an extension function on **List<Job>** called **joinAll**, which will suspend and wait for all jobs in the list to complete for returning. If you keep track of the jobs that are associated with your workers, you can use **joinAll** to wait for them all to finish before closing the `fetchedFlightsChannel`.

Update your worker creation logic to remember your workers' jobs and wait for them to finish before closing the `fetchedFlightsChannel`.

Listing 22.8 Joining Jobs (FlightWatcher.kt)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis",
        "Estragon", "Taernyl"),
    numberOfWorkers: Int = 2
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    val fetchedFlightsChannel = Channel<FlightStatus>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
        passengerNamesChannel.close()
    }

    repeat(numberOfWorkers) {
        launch {
            fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
            fetchedFlightsChannel.close()
        }
    }
    launch {
        (1..numberOfWorkers).map {
            launch {
                fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
            }
        }.joinAll()
        fetchedFlightsChannel.close()
    }

    fetchedFlightsChannel.toList()
}
...
```

Run your application again. All four of your passengers will be fetched, and your simulation will begin, as shown

below. You will see logs indicating that there are multiple network requests in-flight at the same time.

```
Getting the latest flight info...
Started fetching flight info
Started fetching loyalty info
Started fetching flight info
Started fetching loyalty info
Combining flight data
Combining flight data
Finished fetching loyalty info
Finished fetching loyalty info
Finished fetching flight stats
Finished fetching flight stats
Fetched flight: FlightStatus(flightNumber=RX7759, passengerName=Madrigal, ...)
Fetched flight: FlightStatus(flightNumber=UC4790, passengerName=Polarcubis, ...)
...
Fetched flight: FlightStatus(flightNumber=TF3942, passengerName=Taernyl, ...)
Fetched flight: FlightStatus(flightNumber=RD2604, passengerName=Estragon, ...)
Found flights for Madrigal (RX7759), Polarcubis (UC4790), Taernyl (TF3942),
    Estragon (RD2604)
There are 4 flights being tracked
Madrigal: Other passengers are boarding (Flight departs in 28 minutes)
...
Madrigal: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking Madrigal's flight
There are 3 flights being tracked
Polarcubis: Boarding will start soon (Flight departs in 111 minutes)
...
Polarcubis: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking Polarcubis's flight
There are 2 flights being tracked
Taernyl: You can now board the plane (Flight departs in 55 minutes)
...
Taernyl: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking Taernyl's flight
There are 1 flights being tracked
Estragon: Your flight was canceled (Flight departs in 45 minutes)
Finished tracking Estragon's flight
Finished tracking all flights
```

(If a passenger’s flight is canceled or is leaving immediately, you will only see one status message for their flight, followed by the “finished tracking flight” message. Also, you might notice that your passengers are tracked in a slightly different order. The simulation is iterating through passengers in the order they finish fetching, which might be slightly different from the order they were declared in.)

Notice, at the top of your output, that there are two identical logs back to back, indicating that two flights are being fetched in parallel. If you were to time your flight fetching, you would see that it now takes approximately 10 seconds (on a solid network connection), down from 20 seconds if you had not made the changes in this chapter.

This application of **Channels** is called a “fan-out,” because the pipeline gets wider when you go from the producer to the workers. Channels can also be used to create a “fan-in” scenario, where multiple coroutines send to the same channel.

At a high level, channels and flows have some striking similarities. They are both used with coroutines and emit values over time to consumers. But they have different use cases.

Think about what your code would look like if you tried to implement your fan-out logic using a **Flow** instead of a **Channel**. When you send a value to a **Flow**, there are no guarantees about how many observers will receive the value. If no other component is collecting from a **Flow**, then a value might not make its way anywhere. And if there are multiple collectors, they might each get the same value.

This is not ideal for your fan-out implementation, because you want each request to be sent to exactly one worker.

On the other hand, **Channels** would not be a good fit for the flight status streams you set up in [Chapter 21](#). Those flows contained data and state related to your application. If multiple parts of your app need to be aware of the same data – like Madrigal’s boarding information – then you want several collectors to each receive the same values. And sometimes – like if Madrigal has navigated to a different screen that does not show her current flight – this information might not be needed, and you might not have any collectors for your **Flow**.

You will likely find yourself reaching for **Flow** more often than **Channel**, but both tools are invaluable. Channels are the perfect fit when you have two independent coroutines that need to communicate safely. For virtually all other observable streams of data and for modeling application state, go with flow.

TaernylAir is now complete. Passengers have all the information they need to board their flights, and users tracking multiple flights can now do so without waiting too long for flight data.

In the next part of the book, you will learn about Kotlin’s ability to interoperate with Java code, making it possible to introduce Kotlin into an existing Java project. You will also learn about Kotlin’s multiplatform techniques, which allow you to use Kotlin code beyond the JVM.

For the More Curious: Other Channel Behaviors

Channels come in multiple flavors, and the type of channel determines how the **send** and **receive** functions behave.

Rendezvous channels

When you call the **Channel** constructor with no arguments, you get the default [rendezvous channel](#). This is the kind of channel you used in this chapter.

A rendezvous channel is a channel that has no buffer. When you send an item into a rendezvous channel, the **send** function suspends until a receiver calls **receive** from the channel to obtain the value. Similarly, if you call **receive** before **send**, your code will suspend until a value is sent to the channel.

(Why “rendezvous”? The **send** and **receive** calls wait for each other to meet up – a rendezvous, if you will.)

Buffered channels

When you create a channel, you can also specify a buffer size. This gets you a [buffered channel](#).

There are two ways of creating a buffered channel, both shown below:

```
val defaultBufferedChannel = Channel(BUFFERED)

val bufferSize = 5
val bufferedChannel = Channel(bufferSize)
```

The first example uses the runtime-default buffer size for a **Channel** (generally 64). The second uses a custom buffer size, where the provided **Int** specifies how many items can be placed in the buffer.

With a buffered channel, if the buffer is not full and you call **send**, the value will be placed into the buffer, and the **send** call does not need to suspend. If you call **send** when the buffer is full, then you get the same behavior as a rendezvous channel: The **send** call will suspend until a value is removed from the buffer with **receive**.

The buffer acts as a first-in-first-out (or “FIFO”) queue. If a value is in the buffer, a receiver will immediately get the oldest value in the buffer, without suspending. If the buffer is empty, a **receive** call will suspend until a value is sent to the channel.

Unlimited channels

[Unlimited channels](#) are buffered channels that do not limit the size of the buffer. As long as your program has access to enough memory, items can be added to the channel’s buffer. If memory is not available, your program will crash.

In practice, this means that every **send** call to the channel will complete immediately and not suspend. (The **send** function is still a suspending function and must be called from a coroutine, but it will not actually suspend.)

Receivers behave the same as in a buffered channel: If the buffer is not empty, a receiver will instantly be given the oldest value in the buffer. If the buffer *is* empty, the **receive** call will suspend until a value is sent.

You create an unlimited channel by passing the `Channel.UNLIMITED` constant as an argument to the **Channel** constructor:

```
val unlimitedChannel = Channel(UNLIMITED)
```

Conflated channels

The last type of channel that can be created with the **Channel** constructor is called a [conflated channel](#). A conflated channel is a channel that can buffer one item and replaces the item in the buffer with new values instead of suspending. This can be helpful when a new value should supersede a value that might already be in the buffer – for example, if you have a channel that sends events in response to the user clicking a button in your program, you might only care about the most recent button click.

As in an unlimited channel, the **send** function never suspends. The **receive** function behaves the same as it does for the other buffered channels: If a value is in the buffer, then it is used and removed from the buffer. (Because the buffer holds a single value, any value held is by definition the oldest value.) If the buffer does not have a value, **receive** suspends until **send** is called.

You get a conflated channel by using the `Channel.CONFLATED` constant as an argument to the **Channel** constructor:

```
val conflatedChannel = Channel(CONFLATED)
```

There are other arguments that you can experiment with when creating a channel. For channels with a fixed-size buffer (excluding conflated channel), you can specify alternative behavior for what should happen when the buffer is full.

As an example, consider a buffered channel used to process a user's keystrokes or mouse movements. If events are sent to the channel more quickly than they can be processed, you can specify a strategy for what to do with the events when the buffer fills. One option would be the `BufferOverflow.DROP_LATEST` argument, which causes events to not be sent into a channel if its buffer is full.

In general, a rendezvous channel or a buffered channel will suit the majority of your channel needs, especially given the ability to customize a buffered channel.

Part VI

Interoperation and Multiplatform Applications

Throughout the book, you have been using Kotlin/JVM to write Kotlin code that runs in the Java Virtual Machine. As we mentioned in the introduction of this book, Kotlin can be used outside of the JVM. Kotlin/JS lets you write Kotlin code for the web, and Kotlin/Native lets your Kotlin code run natively on platforms like iOS, macOS, Windows, Linux, and more.

In this part of the book, we will discuss how Kotlin's interoperability features work across its various targets. Initially, you will learn about Kotlin's Java interoperability features, which can help you introduce Kotlin into an existing codebase. Then you will write a multiplatform application that targets the JVM, macOS desktop (natively), and web using a single codebase that shares Kotlin code between platforms.

Java Interoperability

There are many reasons to learn Kotlin. For Java developers, for example, Kotlin offers a more modern and safe language for existing projects. If that is you, we hope you are inspired to use Kotlin to improve your Java projects.

Many other developers who learn Kotlin will use it in projects that are written 100% with Kotlin. But even for these developers, the need to interoperate with Java still comes up. You may work with a framework that was written in Java, for example, or use a library that has expectations about the structure of your program's output. Being aware of these concerns can be extremely helpful when getting yourself out of tricky situations.

Up to this point in the book, you have been using Kotlin/JVM, which means that your code gets compiled into Java bytecode. Because this bytecode is no different than the bytecode that gets generated from regular Java code, Kotlin has features that make it [interoperable](#) with Java – that is, it functions alongside and works with Java code.

This is likely the most important feature of the Kotlin programming language. Full interoperability with Java means that Kotlin files and Java files can exist in the same project, side by side. You can invoke Java methods from Kotlin and Kotlin functions from Java. You can also use existing Java libraries and frameworks from Kotlin (two noteworthy examples are Android and Spring).

Full interoperability with Java also means that you can slowly transition your codebase from Java to Kotlin. Maybe you do not have the opportunity to rebuild your project entirely in Kotlin – consider moving new feature development to Kotlin. Perhaps you would like to convert the Java files in your application that will see the most benefit from a move to Kotlin – consider converting your model objects or your unit tests.

This chapter will show you how Java and Kotlin files interoperate and discuss the things you should consider when writing code that will be used from both Kotlin and Java.

Interoperating with a Java Class

For this chapter, create a new project in IntelliJ called Interop. Use the same steps you have used for other projects in this book, selecting the Application template as before and setting the Project JDK from the dropdown. This template has the support you will need for declaring Java code alongside your Kotlin code.

Interop will contain two files: `Hero.kt`, a Kotlin file that represents the hero from NyetHack, and `Jhava.java`, a Java class that represents a monster from another realm.

In this chapter, you will write both Kotlin code and Java code. If you do not have experience writing Java code, fear not, as the Java in these examples should be intuitive given your Kotlin experience.

Get started by creating a folder to host Java code. Locate your `main` directory in IntelliJ’s project pane (it is nested in `src`). Right-click the `main` directory and select `New → Directory`. Name the directory `java`. (IntelliJ will suggest this name, which you can double-click to accept.)

Java code must appear in the `java` folder that you just created. If you place Java code in the default `kotlin` folder, it will be ignored and will not be compiled as part of your project. (However, Kotlin code *can* appear in your `java` folder with no change in behavior. This feature can be useful for developers wanting to gradually transition to Kotlin from Java.)

With your new `java` folder in place, right-click it and select `New → Java Class`. When prompted, name your new class `Jhava`. In the **`Jhava`** class, define a method called **`utterGreeting`** that returns a **`String`**:

Listing 23.1 Declaring a class and method in Java (`Jhava.java`)

```
public class Jhava {
    public String utterGreeting() {
        return "BLARGH";
    }
}
```

Now, create a new Kotlin file called `Hero.kt` in `src/main/kotlin`. Give it a **`main`** function and declare an adversary `val`, an instance of **`Jhava`**:

Listing 23.2 Declaring a main function and Jhava adversary in Kotlin (`Hero.kt`)

```
fun main() {
    val adversary = Jhava()
}
```

That is it! You have written a line of Kotlin code that instantiates a Java object – crossing the barrier between the two languages. Java interoperability in Kotlin really is that easy.

But we do have more to show you, so let’s press on. As a test, print out the greeting that the **`Jhava`** adversary utters.

Listing 23.3 Invoking a Java method in Kotlin (`Hero.kt`)

```
fun main() {
    val adversary = Jhava()
    println(adversary.utterGreeting())
}
```

You have now instantiated a Java object and invoked a Java method on it, all from Kotlin. Run `Hero.kt`. You should see the monster’s greeting (BLARGH) printed out to the console.

Kotlin was created to interoperate seamlessly with Java. It was also created with a number of improvements over Java. Do you have to give up the improvements when you want to interoperate? Not at all. With some awareness of the differences in the two languages and the help of annotations available on each side, you can enjoy the best of what Kotlin has to offer.

Interoperability and Nullity

Add another method to **Jhava** called **determineFriendshipLevel**. It should return a value of type **String** and, because the monster does not understand friendship, a value of null.

Listing 23.4 Returning null from a Java method (Jhava.java)

```
public class Jhava {
    public String utterGreeting() {
        return "BLARGH";
    }

    public String determineFriendshipLevel() {
        return null;
    }
}
```

Call this new method from **Hero.kt**, storing the monster’s friendship level in a **val**. You are going to print this value out to the console, but, remembering that the monster yelled its greeting at you in all caps, go ahead and lowercase the friendship level before printing it out.

Listing 23.5 Printing the friendship level (Hero.kt)

```
fun main() {
    val adversary = Jhava()
    println(adversary.utterGreeting())

    val friendshipLevel = adversary.determineFriendshipLevel()
    println(friendshipLevel.lowercase())
}
```

Run **Hero.kt**. Although the compiler did not alert you to any problems, the program crashes at runtime:

```
BLARGH
Exception in thread "main" java.lang.NullPointerException: friendshipLevel must
not be null
    at HeroKt.main(Hero.kt:6)
    at HeroKt.main(Hero.kt)
```

In Java, all objects can be null. When you call a Java method like **determineFriendshipLevel**, the API seems to advertise that the method will return a **String**, but that does not mean that you can assume that the return value will play by Kotlin’s rules about nullity.

Because all objects in Java can be null, it is safer to assume that values are nullable unless otherwise specified. However, while this assumption is safer, it can lead to considerably more verbose code, as you will have to handle the nullability of each and every Java variable you reference.

In **Hero.kt**, move your text caret to **friendshipLevel** and press Control-Shift-P to reveal its type. IntelliJ reports that the method returns a value of type **String!**. The exclamation mark means that the return value could be either **String** or **String?**. The Kotlin compiler does not know whether the value of the string being returned from Java is null.

These ambiguous return value types are called [platform types](#). Platform types are not syntactically meaningful; they are only displayed in the IDE and in other documentation. You also cannot define a platform type in your own Kotlin code. It only exists as an interoperability mechanism.

Platform types can be difficult to work with, because they hide the actual nullability of the value in question. Fortunately, authors of Java code can write Kotlin-friendly code that advertises nullity more explicitly using nullability annotations. Explicitly declare that **determineFriendshipLevel** can return a value of null by adding a **Nullable** annotation to its method header.

Listing 23.6 Specifying that a return value will possibly be null (Jhava.java)

```
public class Jhava {
    public String utterGreeting() {
        return "BLARGH";
    }

    @Nullable
    public String determineFriendshipLevel() {
        return null;
    }
}
```

(You will need to import `org.jetbrains.annotations.Nullable`, which IntelliJ will offer to do for you.)

Nullable warns the consumer of this API that the method can return null (not that it *must* return null). The Kotlin compiler recognizes this annotation. Return to **Hero.kt** and note that IntelliJ is now warning you about invoking **lowercase** directly on a **String?**.

Replace this direct invocation with a safe call.

Listing 23.7 Handling nullability with the safe call operator (Hero.kt)

```
fun main() {
    val adversary = Jhava()
    println(adversary.utterGreeting())

    val friendshipLevel = adversary.determineFriendshipLevel()
```



```
        println(friendshipLevel?.lowercase())
    }
```

Run `Hero.kt`. Now, `null` should be printed to the console.

Because `friendshipLevel` is `null`, you may want to provide a default friendship level. Use the null coalescing operator to provide a default to be used when `friendshipLevel` is `null`.

Listing 23.8 Providing a default value with the Elvis operator (`Hero.kt`)

```
fun main() {
    val adversary = Jhava()
    println(adversary.utterGreeting())

    val friendshipLevel = adversary.determineFriendshipLevel()
    println(friendshipLevel?.lowercase() ?: "It's complicated")
}
```

Run `Hero.kt`, and you should see `It's complicated`.

You used `Nullable` to signify that a method could return `null`. You can specify that a value will definitely not be `null` using the `NotNull` annotation. This annotation is nice, because it means that the consumer of this API does not need to worry that the value returned could be `null`. The **Jhava** monster's greeting should not be `null`, so add a `NotNull` annotation to the **utterGreeting** method header.

Listing 23.9 Specifying that a return value will not be `null` (`Jhava.java`)

```
public class Jhava {

    @NotNull
    public String utterGreeting() {
        return "BLARGH";
    }

    @Nullable
    public String determineFriendshipLevel() {
        return null;
    }
}
```

(Again, you will need to import the annotation.)

Nullability annotations can be used to add context to return values, parameters, and even fields.

Kotlin provides a variety of tools for dealing with nullability, including prohibiting normal types from being `null`. If you write Kotlin code, then the most common source of issues with `null` is interoperation, so take care when calling Java code from Kotlin.

Type Mapping

Kotlin’s types often correspond one to one with Java types. A **String** in Kotlin is a **String** when compiled down to Java. This means that a **String** returned from Java methods can be used in the same way in Kotlin as a **String** explicitly declared in Kotlin.

But there are some type mappings that are not one to one between Kotlin and Java. For an example, consider basic data types. As we discussed in [the section called *For the More Curious: Java Primitive Types in Kotlin*](#) in [Chapter 2](#), Java represents basic data types using what it calls *primitive types*. Primitive types are not objects in Java, but all types are objects in Kotlin – including basic data types. However, the Kotlin compiler maps Java primitives onto the most similar Kotlin type.

To see type mapping in action, add an integer called `hitPoints` to **Jhava**. An integer is represented by the object type **Int** in Kotlin and by the primitive type **int** in Java.

Listing 23.10 Declaring an int in Java (Jhava.java)

```
public class Jhava {  
    public int hitPoints = 52489112;  
  
    @NotNull  
    public String utterGreeting() {  
        return "BLARGH";  
    }  
  
    @Nullable  
    public String determineFriendshipLevel() {  
        return null;  
    }  
}
```

Now, obtain a reference to `hitPoints` in `Hero.kt`.

Listing 23.11 Referencing a Java field from Kotlin (Hero.kt)

```
fun main() {  
    val adversary = Jhava()  
    println(adversary.utterGreeting())  
  
    val friendshipLevel = adversary.determineFriendshipLevel()\  
    println(friendshipLevel?.lowercase() ?: "It's complicated")  
  
    val adversaryHitPoints: Int = adversary.hitPoints  
}
```

Although `hitPoints` is defined in the **Jhava** class as an **int**, you refer to it here as an **Int** with no problem. (You are not using type inference here only to illustrate the type mapping. Explicit type declarations are not required for interoperability: `val adversaryHitPoints = adversary.hitPoints` would work just as well, and the inferred type would still be **Int**.)

Now that you have a reference to this integer, you can invoke functions on it. Ensure that the monster does not have more than 100 hit points with the **coerceAtMost** function.

Listing 23.12 Invoking a function on a Java field from Kotlin (Hero.kt)

```
fun main() {  
    ...  
    val adversaryHitPoints: Int = adversary.hitPoints  
    println(adversaryHitPoints.coerceAtMost(100))  
}
```

Run `Hero.kt` to print out the adversary’s hit points. You will see 100 printed to the console.

In Java, methods cannot be invoked on primitive types. In Kotlin, the integer `adversaryHitPoints` is an object of type **Int**, and functions can be called on that **Int**.

As another illustration of type mapping, print the name of the Java class backing `adversaryHitPoints`.

Listing 23.13 Printing Java backing class name (Hero.kt)

```
fun main() {  
    ...  
    val adversaryHitPoints: Int = adversary.hitPoints  
    println(adversaryHitPoints.coerceAtMost(100))  
    println(adversaryHitPoints.javaClass)  
}
```

When you run `Hero.kt`, you will see `int` printed to the console. Although you can invoke **Int** functions on `adversaryHitPoints`, the variable is a primitive **int** at runtime. As you may recall from the bytecode you looked at in [Chapter 2](#), all mapped types are mapped back to their Java counterparts at runtime. Kotlin gives you the power of objects when you want them and the performance of primitive types when you need them.

Getters, Setters, and Interoperability

Kotlin and Java handle class-level variables quite differently. Java uses fields and typically gates access via accessor and mutator methods. Kotlin, as you have seen, features properties, which restrict access to backing fields and may automatically expose accessors and mutators.

In the last section, you added a public `hitPoints` field to **Jhava**. This worked to illustrate type mapping, but it violates the principle of encapsulation – so is not a good solution. In Java, fields should be accessed or mutated using getter and setter methods. Getters can be used to access data, and setters can be used to mutate data.

Make `hitPoints` private and create a getter method so `hitPoints` can be accessed but not mutated.

Listing 23.14 Declaring a field in Java (Jhava.java)

```
public class Jhava {  
    public private int hitPoints = 52489112;  
  
    @NotNull  
    public String utterGreeting() {  
        return "BLARGH";  
    }  
  
    @Nullable  
    public String determineFriendshipLevel() {  
        return null;  
    }  
  
    public int getHitPoints() {  
        return hitPoints;  
    }  
}
```

Now return to `Hero.kt`. Note that your code still compiles. Recall from [Chapter 13](#) that Kotlin generates getters and setters, but the syntax to use them looks like you are directly accessing a variable on the class.

Although you could directly call the **`getHitPoints`** method, it is preferred to use the property access syntax you are already using. This lets you use the same syntax that you would for a Kotlin class while still maintaining encapsulation. Because **`getHitPoints`** is prefixed with `get`, you can drop the prefix in Kotlin and refer to it simply as `hitPoints`. This Kotlin feature transcends the barrier between Kotlin and Java.

The same goes for setters. By now your hero and the **Jhava** monster are well acquainted and wish to communicate further. The hero would like to expand the monster’s vocabulary beyond a single utterance. Pull the monster’s greeting out into a field and add a getter and a setter so the hero can modify the greeting in an attempt to teach the monster language.

Listing 23.15 Exposing a greeting in Java (Jhava.java)

```
public class Jhava {  
    private int hitPoints = 52489112;  
    private String greeting = "BLARGH";  
  
    @NotNull  
    public String utterGreeting() {  
        return "BLARGH" + greeting;  
    }  
  
    public String getGreeting() {  
        return greeting;  
    }  
  
    public void setGreeting(String greeting) {  
        this.greeting = greeting;  
    }  
    ...  
}
```

In `Hero.kt`, modify `adversary.greeting`.

Listing 23.16 Setting a Java field from Kotlin (Hero.kt)

```
fun main() {  
    ...  
    val adversaryHitPoints: Int = adversary.hitPoints  
    println(adversaryHitPoints.coerceAtMost(100))  
    println(adversaryHitPoints.javaClass)  
  
    adversary.greeting = "Hello, Hero."  
    println(adversary.utterGreeting())  
}
```

You can use assignment syntax to mutate a Java field, rather than calling its associated setter. You have the syntax benefits provided in Kotlin, even while working with Java APIs. There is one slight caveat for setters, though: This automatic conversion works for all getters with the `get` prefix, but setters must have a corresponding getter to be eligible for this treatment. If you only have a setter for a field and not a getter, you cannot use property access syntax.

Run `Hero.kt` to see that the hero has taught the **Jhava** monster some language.

Beyond Classes

Kotlin affords developers greater flexibility with respect to the format of the code they write. A Kotlin file can include classes, functions, and variables – all at the top level of the file. In Java, a file represents exactly one class. How, then, are top-level functions declared in Kotlin represented in Java?

Expand the interspecies communication with a proclamation from the hero. Declare a function called **makeProclamation** in `Hero.kt` outside of **main**.

Listing 23.17 Declaring a top-level function in Kotlin (Hero.kt)

```
fun main() {  
    ...  
}  
  
fun makeProclamation() = "Greetings, beast!"
```

You will need a way to invoke this function from Java, so add a **main** method to **Jhava**.

Listing 23.18 Defining a main method in Java (Jhava.java)

```
public class Jhava {  
  
    private int hitPoints = 52489112;  
    private String greeting = "BLARGH";  
  
    public static void main(String[] args) {  
  
    }  
    ...  
}
```

In that **main** method, print out the value returned by **makeProclamation**, referencing the function as a static method in the class **HeroKt**:

Listing 23.19 Referencing a top-level Kotlin function from Java (Jhava.java)

```
public class Jhava {  
    ...  
    public static void main(String[] args) {  
        System.out.println(HeroKt.makeProclamation());  
    }  
    ...  
}
```

Top-level functions defined in Kotlin are represented as static methods in Java and are called as such.

makeProclamation is defined in `Hero.kt`, so the Kotlin compiler creates a class called **HeroKt** for the static method to be associated with.

If you would like `Hero.kt` and `Jhava.java` to interoperate a bit more fluidly, you can change the name of the generated class with the `@file:JvmName` annotation. Do this at the top of `Hero.kt`:

Listing 23.20 Specifying compiled class name using @file:JvmName (Hero.kt)

```
@file:JvmName("Hero")  
  
fun main() {  
    ...  
}  
  
fun makeProclamation() = "Greetings, beast!"
```

Now, in **Jhava**, you can reference the **makeProclamation** function more cleanly.

Listing 23.21 Referencing a renamed top-level Kotlin function from Java (Jhava.java)

```
public class Jhava {  
    ...  
    public static void main(String[] args) {  
        System.out.println(HeroKt.makeProclamation());  
    }  
    ...  
}
```

Run `Jhava.java` (using the run button in the editor’s gutter, since your saved run configuration is for `Hero.kt`) to read your hero’s proclamation. Annotations like `@file:JvmName` give you direct control over what Java code is generated when you write Kotlin code.

Another important JVM annotation is `@JvmOverloads`. Kotlin’s default parameters empower you to replace verbose, repetitive method overloading with a streamlined approach to providing options in your API. What does this mean in practice? The example below should clarify things.

Add a new function called **handOverFood** to `Hero.kt`.

Listing 23.22 Adding a function with default parameters (Hero.kt)

```
...  
fun makeProclamation() = "Greetings, beast!"  
  
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {  
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")  
}
```

The hero offers some food in the **handOverFood** function, and the function’s caller has options for invoking it due to its default parameters. The caller can specify what is in the hero’s left hand and/or right hand – or accept the default options of berries and beef. Kotlin gives the caller options without adding much complexity to the code.

Java, on the other hand, which lacks default parameters, would accomplish this with method overloading:

```
public static void handOverFood(String leftHand, String rightHand) {
    System.out.println("Mmmm... you hand over some delicious " +
        leftHand + " and " + rightHand + ".");
}

public static void handOverFood(String leftHand) {
    handOverFood(leftHand, "beef");
}

public static void handOverFood() {
    handOverFood("berries", "beef");
}
```

Method overloading in Java requires much more code than default parameters in Kotlin. Also, one option for calling the Kotlin function cannot be replicated in Java – the option of using the default value for the first parameter, `leftHand`, while passing a value for the second parameter, `rightHand`. Kotlin’s named function arguments make this option possible: `handOverFood(rightHand = "cookies")` will result in `Mmmm... you hand over some delicious berries and cookies..` But Java does not support named method parameters, so it has no way to distinguish between methods called with the same number of parameters (unless the parameters are of different types).

As you will see in a moment, the `@JvmOverloads` annotation triggers the generation of the three corresponding Java methods so that Java consumers are, for the most part, not left out.

The **Jhava** monster abhors fruit. It would like to be offered pizza or beef instead of berries. Add a method called **offerFood** to `Jhava.java` that exposes a way for a **Hero** to offer food to a **Jhava** monster.

Listing 23.23 Only one method signature (Jhava.java)

```
public class Jhava {
    ...
    public int getHitPoints() {
        return hitPoints;
    }

    public void offerFood() {
        Hero.handOverFood("pizza");
    }
}
```

This call to **handOverFood** causes a compiler error, because Java has no concept of default method parameters. As such, a version of **handOverFood** with only one parameter does not exist in Java. To verify, take a look at the decompiled Java bytecode for **handOverFood**:

```
public static final void handOverFood(@NotNull String leftHand,
    @NotNull String rightHand) {
    Intrinsic.checkNotNullParameter(leftHand, "leftHand");
    Intrinsic.checkNotNullParameter(rightHand, "rightHand");
    String var2 = "Mmmm... you hand over some delicious " +
        leftHand + " and " + rightHand + '.';
    boolean var3 = false;
    System.out.println(var2);
}
```

While you have the option to avoid method overloading in Kotlin, your Java counterparts are not afforded the same luxury. The `@JvmOverloads` annotation will help your API consumers in Java by providing overloaded versions of your Kotlin function. Add the annotation to **handOverFood** in `Hero.kt`.

Listing 23.24 Adding @JvmOverloads (Hero.kt)

```
...
fun makeProclamation() = "Greetings, beast!"

@JvmOverloads
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}
```

Your invocation of **handOverFood** in **Jhava.offerFood** no longer causes an error, because it is now calling a version of **handOverFood** that exists in Java. You can again confirm this by looking at the new decompiled Java bytecode:

```
@JvmOverloads
public static final void handOverFood(@NotNull String leftHand,
    @NotNull String rightHand) {
    Intrinsic.checkNotNullParameter(leftHand, "leftHand");
    Intrinsic.checkNotNullParameter(rightHand, "rightHand");
    String var2 = "Mmmm... you hand over some delicious " +
        leftHand + " and " + rightHand + '.';
    boolean var3 = false;
    System.out.println(var2);
}

@JvmOverloads
public static final void handOverFood(@NotNull String leftHand) {
    handOverFood$default(leftHand, (String)null, 2, (Object)null);
}

@JvmOverloads
public static final void handOverFood() {
    handOverFood$default((String)null, (String)null, 3, (Object)null);
}
```

The single-parameter method specifies the first parameter from the Kotlin function: `leftHand`. When this method is called, the default value for the second parameter will be used.

To test offering food to the monster, call **offerFood** in `Hero.kt`:

Listing 23.25 Testing out offerFood (Hero.kt)

```
@file:JvmName("Hero")

fun main() {
    ...
    adversary.greeting = "Hello, Hero."
    println(adversary.utterGreeting())

    adversary.offerFood()
}

fun makeProclamation() = "Greetings, beast!"
...
```

Run `Hero.kt` to confirm that the hero hands over pizza and beef.

If you are designing an API that may be exposed to Java consumers, consider using `@JvmOverloads` for an API that is nearly as robust for Java developers as it is for Kotlin developers.

There are two more JVM annotations that you should consider when writing Kotlin code that will interoperate with Java code, and they both have to do with classes. `Hero.kt` does not yet have a class implementation, so add a new class called **Spellbook**. Give **Spellbook** one property, `spells` – a list of string spell names.

Listing 23.26 Declaring the Spellbook class (Hero.kt)

```
...
@JvmOverloads
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}

class Spellbook {
    val spells = listOf("Magic Ms. L", "Lay on Hans")
}
```

Again, Kotlin and Java handle class-level variables quite differently: Java uses fields with getter and setter methods, while Kotlin has properties with backing fields. As a result, while in Java you can access a field directly, in Kotlin you will be routed through an accessor – even though the access syntax may be identical.

So, referencing `spells`, a property of **Spellbook**, in Kotlin would look like this:

```
val spellbook = Spellbook()
val spells = spellbook.spells
```

And in Java, accessing `spells` would look like this:

```
Spellbook spellbook = new Spellbook();
List<String> spells = spellbook.getSpells();
```

In Java, calling **getSpells** would be necessary because you cannot directly access the `spells` field. However, you can apply the `@JvmField` annotation to a Kotlin property to expose its backing field to Java consumers and avoid the need for a getter method. Apply `JvmField` to `spells` to expose it directly to **Jhava**:

Listing 23.27 Applying the @JvmField annotation (Hero.kt)

```
...
@JvmOverloads
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}

class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L", "Lay on Hans")
}
```

Now, in `Jhava.java`'s **main** method, you can access `spells` directly to print out each spell:

Listing 23.28 Accessing a Kotlin field directly in Java (Jhava.java)

```
...
public static void main(String[] args) {
    System.out.println(Hero.makeProclamation());

    System.out.println("Spells:");
    Spellbook spellbook = new Spellbook();
    for (String spell : spellbook.spells) {
        System.out.println(spell);
    }
}
...
```

Run `Jhava.java` to confirm that the spells in the spellbook are printed out to the console.

You can also use `@JvmField` to statically represent values in a companion object. Recall from [Chapter 16](#) that companion objects are declared within another class declaration and initialized either when their enclosing class is initialized or when any of their properties or functions are accessed. Add a companion object containing one value, `MAX_SPELL_COUNT`, to **Spellbook**.

Listing 23.29 Adding a companion object to Spellbook (Hero.kt)

```
...
class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L", "Lay on Hans")

    companion object {
        var maxSpellCount = 10
    }
}
```

Now attempt to access `maxSpellCount` from **Jhava**’s **main** method using Java’s static access syntax.

Listing 23.30 Accessing a static value in Java (Jhava.java)

```
public static void main(String[] args) {
    System.out.println(Hero.makeProclamation());

    System.out.println("Spells:");
    Spellbook spellbook = new Spellbook();
    for (String spell : spellbook.spells) {
        System.out.println(spell);
    }

    System.out.println("Max spell count: " + Spellbook.maxSpellCount);
}
...
```

The code does not compile. Why not? When referencing members of a companion object from Java, you must access them first by referencing the companion object and using its accessor:

```
System.out.println("Max spell count: " +
    Spellbook.Companion.getMaxSpellCount());
```

`@JvmField` takes care of all this for you. Add a `@JvmField` annotation to `maxSpellCount` in **Spellbook**’s companion object.

Listing 23.31 Adding the @JvmField annotation to the member of a companion object (Hero.kt)

```
...
class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L", "Lay on Hans")

    companion object {
        @JvmField
        var maxSpellCount = 10
    }
}
```

Once that annotation is in place, your code in `Jhava.java` will compile, because you can access `maxSpellCount` just like any other field in Java. Run `Jhava.kt` to confirm that the maximum spell count is printed to the console.

Although Kotlin and Java handle field access in different ways by default, `@JvmField` is a useful way to expose fields and ensure equivalent access to your Java counterparts.

Functions defined on companion objects run into similar issues when accessed from Java – they have to be accessed via a reference to the companion object. The `@JvmStatic` annotation works like `@JvmField` to allow direct access to functions defined on companion objects. Define a function on **Spellbook**’s companion object called **getSpellbookGreeting**. **getSpellbookGreeting** returns a function to be invoked when **getSpellbookGreeting** is called.

Listing 23.32 Using @JvmStatic on a function (Hero.kt)

```
...
class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L", "Lay on Hans")

    companion object {
        @JvmField
        var maxSpellCount = 10

        @JvmStatic
        fun getSpellbookGreeting() = println("I am the Great Grimoire!")
    }
}
```

Now, invoke **getSpellbookGreeting** in `Jhava.java`.

Listing 23.33 Invoking a static method in Java (Jhava.java)

```
...
public static void main(String[] args) {
    System.out.println(Hero.makeProclamation());

    System.out.println("Spells:");
    Spellbook spellbook = new Spellbook();
    for (String spell : spellbook.spells) {
        System.out.println(spell);
    }

    System.out.println("Max spell count: " + Spellbook.maxSpellCount);

    Spellbook.getSpellbookGreeting();
}
...
```

Run `Jhava.java` to confirm that the spellbook’s greeting is printed to the console.

Although statics do not exist in Kotlin, many commonly used patterns compile down to static variables and methods. Employing the `@JvmStatic` annotation gives you greater control over how Java developers interface with your code.

Exceptions and Interoperability

The hero has taught the **Jhava** monster language, and the monster will now extend its hand in friendship ... or maybe not. Add a method, **extendHandInFriendship**, to `Jhava.java`.

Listing 23.34 Throwing an exception in Java (Jhava.java)

```
public class Jhava {
    ...
    public void offerFood() {
        Hero.handOverFood("pizza");
    }

    public void extendHandInFriendship() throws Exception {
        throw new Exception();
    }
}
```

Invoke this method in `Hero.kt`:

Listing 23.35 Invoking a method that throws an exception (Hero.kt)

```
@file:JvmName("Hero")

fun main() {
    ...
    adversary.offerFood()

    adversary.extendHandInFriendship()
}
...
```

Run this code, and you will see that a runtime exception is thrown. It is not wise to trust a monster.

Recall that exceptions are unchecked in Kotlin. In calling **extendHandInFriendship**, you called a method that throws an exception. In this instance, you knew that when you called the method. Another time, you might not be so lucky. You should take extra care to understand the Java APIs that you are interfacing with from Kotlin.

Wrap your invocation of the **extendHandInFriendship** method in a try/catch block to thwart the monster’s treachery.

Listing 23.36 Handling exceptions using try/catch (Hero.kt)

```
@file:JvmName("Hero")

fun main() {
    ...
    adversary.offerFood()

    try {
        adversary.extendHandInFriendship()
    } catch (e: Exception) {
        println("Begone, foul beast!")
    }
}
...
```

Run `Hero.kt` to see that the hero deftly avoids the monster’s duplicitous attack.

Calling Kotlin functions from Java requires some additional understanding when it comes to handling exceptions. All exceptions in Kotlin, as we have said, are unchecked. But this is not the case in Java – exceptions can be checked, and they must be handled at the risk of a crash. How does this affect calling a Kotlin function from Java?

To see, add a function to `Hero.kt` called **acceptApology**. It is time to exact revenge on the monster.

Listing 23.37 Throwing an unchecked exception (Hero.kt)

```
...
@JvmOverloads
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}

fun acceptApology() {
    throw IOException()
}

class Spellbook {
    ...
}
```

(You will need to import `java.io.IOException`.)

Now, call **acceptApology** from `Jhava.java`.

Listing 23.38 Throwing an exception in Java (Jhava.java)

```
public class Jhava {
    ...
    public void apologize() {
        try {
            Hero.acceptApology();
        } catch (IOException e) {
            System.out.println("Caught!");
        }
    }
}
```

The **Jhava** monster is clever enough to suspect the hero of trickery and wraps its invocation of **acceptApology** in a try/catch block. But the Java compiler warns you that an **IOException** is never thrown in the contents of

the try block – that is, in **acceptApology**. How can this be? **acceptApology** clearly throws an **IOException**.

Understanding this scenario requires a peek into the decompiled Java bytecode:

```
public static final void acceptApology() {  
    throw (Throwable)(new IOException());  
}
```

You can see that an **IOException** is thrown in this function, but nothing about the function’s signature notifies the caller that an **IOException** should be checked for. When the Java compiler complains that **acceptApology** does not throw an **IOException** when invoked from Java, this is why. It has no idea.

Fortunately, there is an annotation to solve this problem, too: `@Throws`. When you use `@Throws`, you include information about the exception that the function throws. Add a `@Throws` annotation to **acceptApology** to augment its Java bytecode.

Listing 23.39 Using the `@Throws` annotation (Hero.kt)

```
...  
@Throws(IOException::class)  
fun acceptApology() {  
    throw IOException()  
}  
  
class Spellbook {  
    ...  
}
```

Now, look at the resulting decompiled Java bytecode:

```
public static final void acceptApology() throws IOException {  
    throw (Throwable)(new IOException());  
}
```

The `@Throws` annotation adds a `throws` keyword to the Java version of **acceptApology**. Looking back at `Jhava.java`, you should see that you have now satisfied the Java compiler, as it can now recognize that **acceptApology** throws an **IOException** that requires checking.

The `@Throws` annotation smooths over some of the ideological differences between Java and Kotlin with respect to exception checking. If you are writing a Kotlin API that may be exposed to a Java consumer, consider using this annotation so your consumer can properly handle any exception thrown.

Function Types in Java

Function types and lambdas are novel inclusions in the Kotlin programming language whose concise syntax for communicating between components is made possible via the `->` operator. They are not supported in versions of Java prior to Java 8.

So what do function types look like when called from Java? The answer may seem deceptively simple: In Java, your function type is represented by an interface with a name like `FunctionN`, where `N` is the number of arguments taken as parameters.

To see this in practice, add a function type called `translator` to `Hero.kt`. `translator` should take a **`String`**, lowercase it, capitalize the first letter, and print out the result.

Listing 23.40 Defining the `translator` function type (`Hero.kt`)

```
fun main() {
    ...
}

val translator = { utterance: String ->
    println(utterance.lowercase().replaceFirstChar { it.uppercase() })
}

fun makeProclamation() = "Greetings, beast!"
```

`translator` is defined like many of the function types that you saw in [Chapter 8](#). It is of type **`(String) -> Unit`**. What will this function type look like in Java? Store the `translator` instance in a variable in `Jhava`.

Listing 23.41 Storing a function type in a variable in Java (`Jhava.java`)

```
public class Jhava {
    ...
    public static void main(String[] args) {
        ...
        Spellbook.getSpellbookGreeting();

        Function1<String, Unit> translator = Hero.getTranslator();
    }
}
```

(You will need to import `kotlin.Unit`; be sure to choose the option from the Kotlin standard library. You will also need to import `kotlin.jvm.functions.Function1`.)

This function is of type **`Function1<String, Unit>`**. **`Function1`** is the base type because `translator` has exactly one parameter. **`String`** and **`Unit`** are used as type parameters because the type of `translator`'s parameter is **`String`** and it returns the Kotlin type **`Unit`**.

There are 23 of these **`Function`** interfaces, ranging from **`Function0`** to **`Function22`**. Each of them includes one function, **`invoke`**. **`invoke`** is used to call a function type, so any time that you need to call a function type, you call **`invoke`** on it. Call `translator` in `Jhava`:

Listing 23.42 Calling a function type in Java (`Jhava.java`)

```
public class Jhava {
    ...
    public static void main(String[] args) {
        ...
        Function1<String, Unit> translator = Hero.getTranslator();
        translator.invoke("TRUCE");
    }
}
```

Run `Jhava.kt` to confirm that `Truce` is printed to the console.

While function types are useful in Kotlin, be mindful of how they are represented in Java. The concise, fluid syntax that you have grown fond of in Kotlin is quite different when called from Java. If your code is visible to Java classes (as a part of an API, for example), then the more considerate route may be to avoid function types. But if you are comfortable with the more verbose syntax, then Kotlin's function types are indeed available to you in Java.

Interoperability between Kotlin and Java is the foundation of Kotlin's growth trajectory. It lets Kotlin leverage existing frameworks, such as Android and Spring, and interface with legacy codebases, giving you a path to gradually introduce Kotlin in your projects.

Fortunately, interoperation between Kotlin and Java is straightforward, with a few small exceptions. Writing Java-friendly Kotlin code and Kotlin-friendly Java code is a useful skill that will pay dividends for developers intending to use Kotlin/JVM.

In the next chapters, you will learn about Kotlin Multiplatform, a technique that allows you to write Kotlin code once and share it across multiple applications running on different platforms.

Introduction to Kotlin Multiplatform

Throughout this book, your Kotlin code has executed in the JVM. The Android runtime is built on top of the JVM, so your code could also execute on Android phones and tablets.

Many developers use Kotlin/JVM, but that is not the only way to use the language. We have mentioned several times that Kotlin is a multiplatform language. In addition to supporting the JVM, Kotlin can also target native platforms like macOS (without running inside the JVM), iOS, and the web (via JavaScript).

If you are an experienced developer, when you think of cross-platform frameworks you may think of tools like React Native, Flutter, and Xamarin. They all have one thing in common: A single codebase is supposed to compile and run directly on every platform your program supports, down to the UI. This is great in theory, but the reality of these cross-platform approaches is not as idyllic as the claims suggest.

Each of those tools enforces its own UI frameworks and APIs. If you happen to know a native framework already, you must learn an entirely new set of APIs before you can write cross-platform code with one of those tools. And you are at the whim of the framework's maintainers to add support for new features that native applications could otherwise start using immediately. If the cross-platform tool of your choosing does not implement a feature that you need to build your application, you are stuck waiting for official support – or you have to integrate the platform's native APIs yourself.

Plus, cross-platform apps often end up implementing some features natively. So if, for example, you wanted to build an iOS app and an Android app, you could end up writing code for three platforms: the cross-platform framework, an iOS feature, *and* an Android feature. Instead of learning one cross-platform tool, as advertised, or two native platforms, now you have to learn three distinct frameworks!

Kotlin Multiplatform takes a different approach, allowing you to build a single Kotlin project that compiles for multiple platforms.

In the next three chapters, you will explore Kotlin's multiplatform capabilities. First, you will set up a new multiplatform project, which will eventually compile for multiple targets. Then, you will see how to write code for Kotlin/Native and Kotlin/JS and how that differs from the code for Kotlin/JVM.

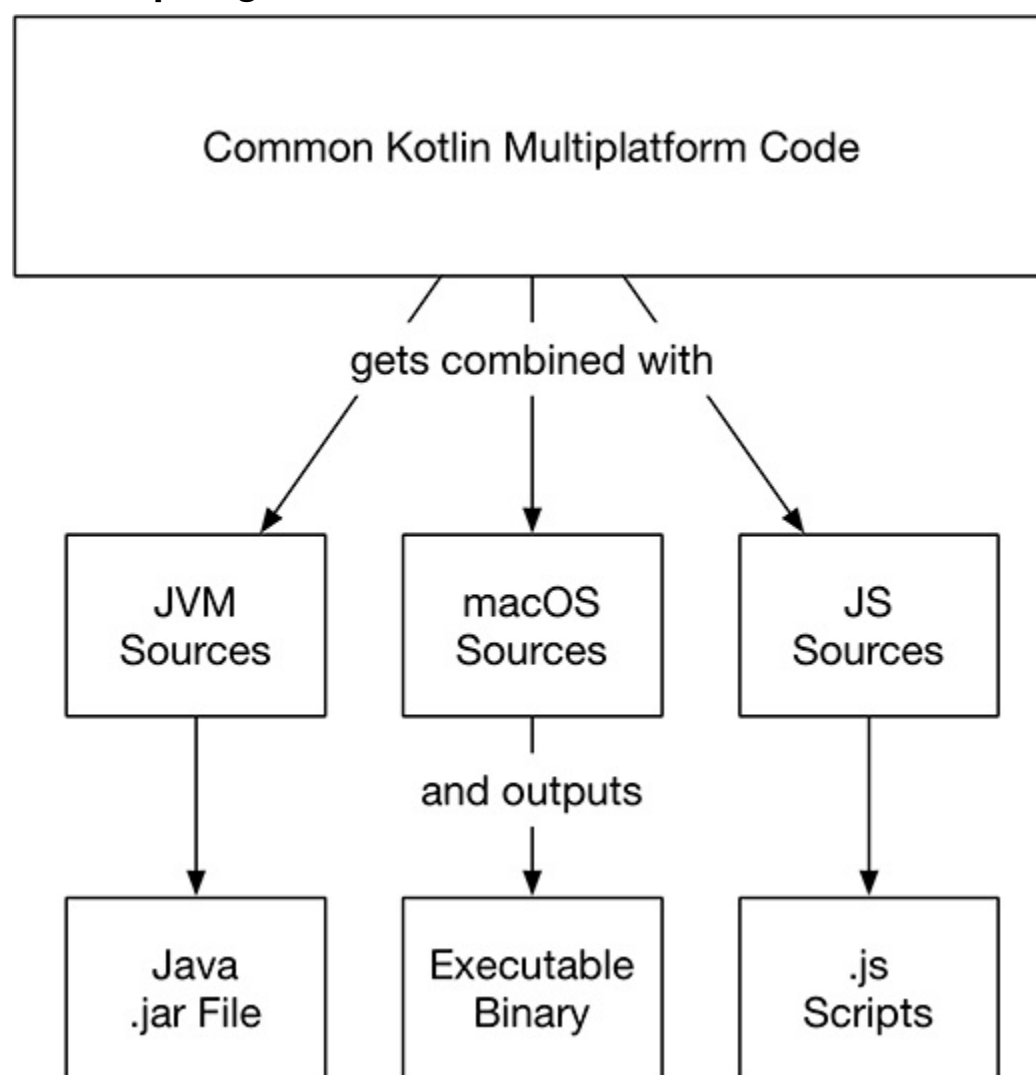
What Is Kotlin Multiplatform?

Kotlin Multiplatform does not attempt to act as a replacement for an existing framework. Instead, it allows you to share Kotlin code across multiple platforms. Codebases that use Kotlin Multiplatform typically have several source code sets (that is, groups of code that are all part of the same project). Your new project will have four source sets:

- common code
- JVM
- macOS
- JavaScript

The majority of your code will live in your common source set. When you compile for a specific target, the compiler will combine the common code with that platform's code. So, for example, if you compile for macOS you will get the common code and the macOS code. The relationship between the source sets and the output binaries is shown in [Figure 24.1](#).

Figure 24.1 Source sets compiling into binaries



We mentioned way back in [Chapter 1](#) that many Kotlin APIs are considered common APIs and work on every target Kotlin supports. The Kotlin Standard Library's API reference shows the platforms any given API supports. You can explore these built-in functions and classes at kotlinlang.org/api/latest/jvm/stdlib/.

Planning Your Multiplatform Project

The first step in making a multiplatform application is to plan what code you want to share. When using Kotlin Multiplatform, we recommend that you focus on sharing code that involves business logic – that is, logic that is integral to your application and its understanding of domain-specific problems. This can include data parsing, complex algorithms, and other pieces of logic that relate to the problem your app is trying to solve.

Although it is technically possible to define entire UIs with Kotlin Multiplatform, we do not recommend it. Each platform you target will likely have a different UI framework, which makes it hard to share UI components between platforms. If you try to build UIs with Kotlin Multiplatform, you will likely end up merely building the UIs in Kotlin, rather than actually *sharing* your UI code.

Instead, we encourage you to declare your UIs in the native language of the platform. This lets you take advantage of all the official UI tooling, which will provide a much better experience than Kotlin can provide and will save you from numerous headaches. Conveniently, Android’s official language is Kotlin, meaning that you will feel at home writing Android UIs.

If you want to go this route of sharing business logic only in a UI-driven application, we recommend that you build a Kotlin Multiplatform *library* rather than an *application*. A library like this is a precompiled binary containing all the APIs you define in your Kotlin Multiplatform project. It cannot be executed on its own and has no **main** function.

Your library can then be imported into the appropriate platform’s project using its official tooling. IntelliJ has a few templates that can set this up, but they are mostly focused on mobile code sharing. You will learn more about sharing code in mobile apps in [Chapter 25](#).

In this chapter, you will build a simple application instead of a library. This will work well for your needs, since you will not be making a complex UI and your application will instead run in a terminal (or a simple web page, in the case of your JavaScript code). However, the principles you will learn apply to both multiplatform applications and libraries, so you will be prepared if you want to go down the library path later on.

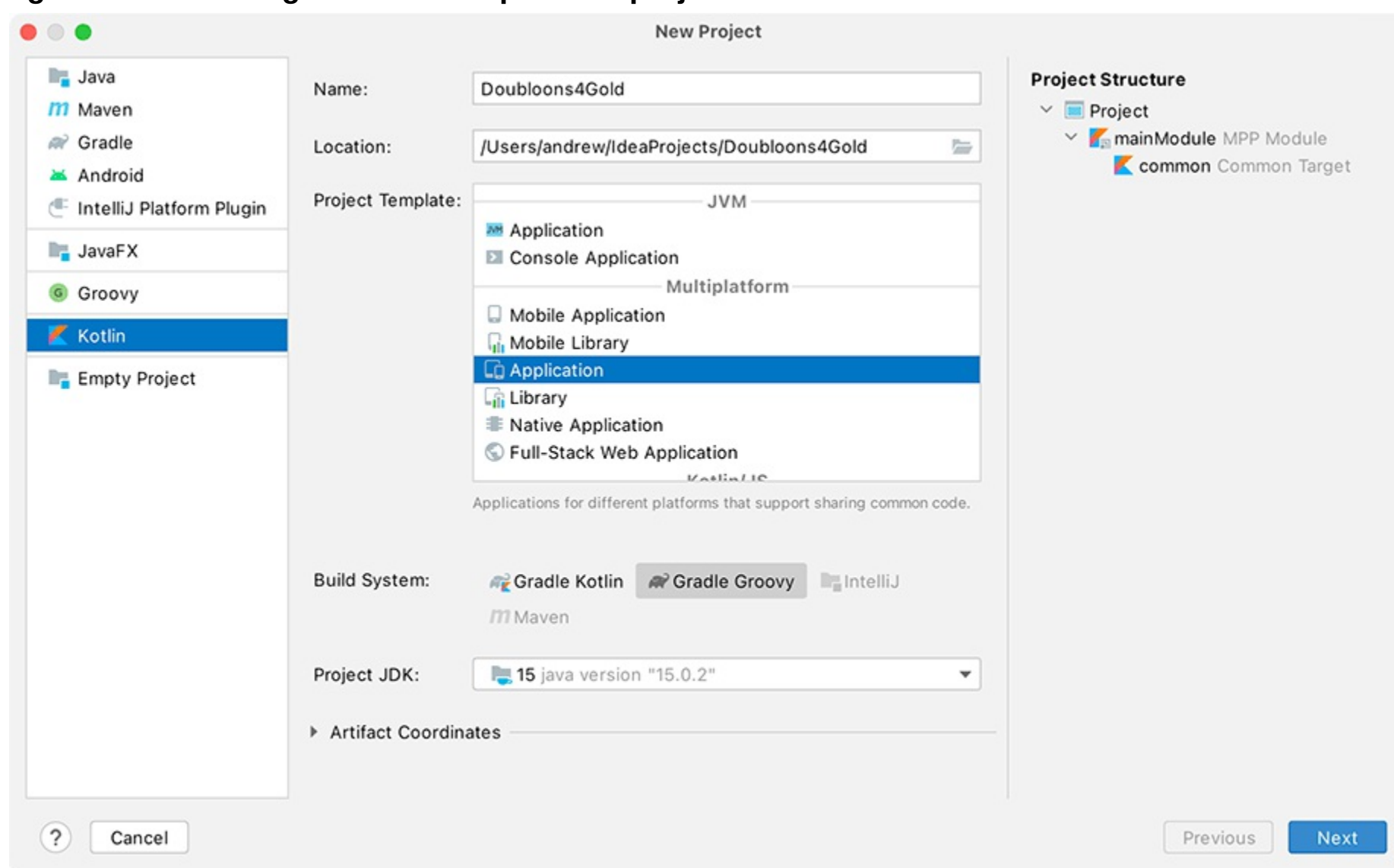
Your First Multiplatform Project

Madrigal has safely landed on a tropical island after her long flight and is ready to relax and enjoy her vacation. But there is one small problem she did not plan for: No one will accept her Kronstadt gold! She will need to convert her gold into doubloons before she can spend any cash.

Exchange rates fluctuate constantly and unpredictably. To manage currency conversion, you will build a multiplatform application that will display the current exchange rate, ask the user how many doubloons they want, and then return the price in gold.

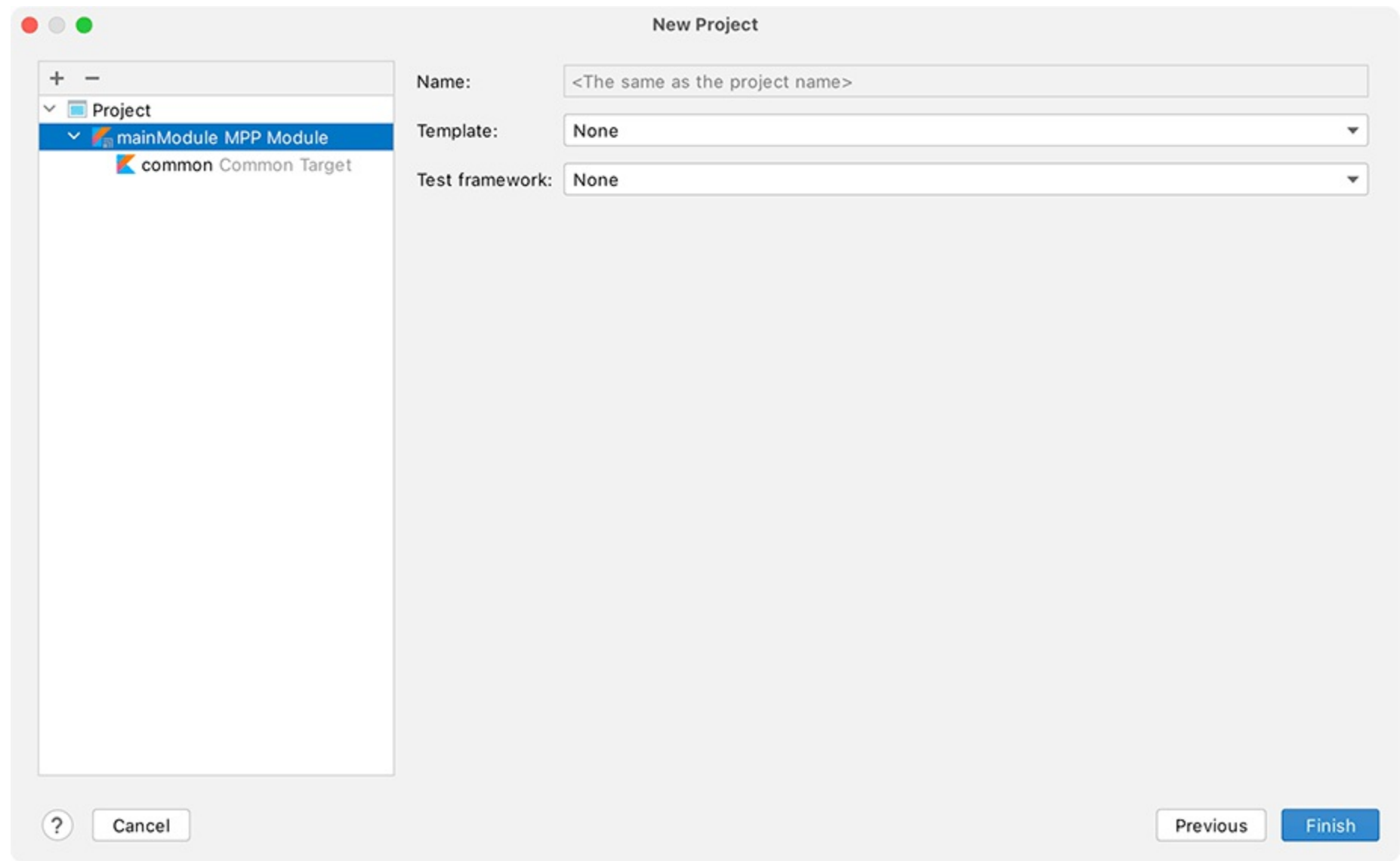
Begin a new project with the New Project wizard, as you have before. But this time, when choosing a template, select Application from the Multiplatform section, as shown in [Figure 24.2](#). As usual, set the Build System to Gradle Groovy and choose a project JDK. Name your new project Doubloons4Gold.

Figure 24.2 Creating a Kotlin Multiplatform project



In the wizard's second window, make sure the Test framework and Template are set to None, as shown in [Figure 24.3](#). Also, notice the project hierarchy on the lefthand side of the screen. This is where you can define targets for your multiplatform application and customize the hierarchy for your project. You should have one entry under mainModule called Common, which is where you will save shared code. For now, leave these options alone. You will define your other targets later.

Figure 24.3 Setting the project template



Click Finish to finalize these options and create your new project.

IntelliJ will perform an initial build of your project, and you will immediately be greeted with a compiler error:

```
Please initialize at least one Kotlin target in 'Doubloons4Gold (:)'.
```

As the error indicates, you need to define a target that your application can compile for.

Defining a Kotlin/JVM Target

The first target that you will add support for is the JVM. This will help you get your feet wet using Kotlin Multiplatform by building on the concepts you have seen before.

Of course, if you *only* wanted to target Kotlin/JVM, you would be better off creating a Kotlin/JVM project, as you have before – not using the Multiplatform template. In the next two chapters, you will define Kotlin/Native and Kotlin/JS targets to make Doubloons4Gold truly multiplatform.

To add a new target to your code, you will need to take a trip to your project’s `build.gradle` file. As you saw in [Chapter 20](#), this file contains information about how to build your project. Previously, you edited this file to declare dependencies for your project. Now, you will update it to tell the Kotlin Multiplatform plugin about the platforms you want to support.

Open your `build.gradle` file. It will look something like this:

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.21'
}

group = 'com.bignerdranch'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

kotlin {
    sourceSets {
        commonMain {
        }
        commonTest {
            dependencies {
                implementation kotlin('test')
            }
        }
    }
}
```

To specify that you want to target a specific platform, add an entry to the **kotlin** block specifying that you want to target the JVM.

Listing 24.1 Targeting the JVM (build.gradle)

```
...
kotlin {
    jvm()
    sourceSets {
        ...
    }
}
```

There are many targets that you can specify for a Kotlin Multiplatform project. [Table 24.1](#) shows some commonly used targets.

Table 24.1 Commonly used Kotlin Multiplatform targets

Platform	Definition in <code>build.gradle</code>	Description
JVM	<code>jvm()</code>	For all systems running Java and Android devices
Android	<code>android()</code>	For Android apps running under the Android runtime with Java
iOS	<code>iosArm64()</code>	For 64-bit ARM iOS devices, including iPhone 5s and newer
iOS simulator	<code>iosX64()</code>	Used by the iOS simulator when running on a Mac
macOS computers	<code>macosX64()</code>	For Intel-based macOS computers ^a
JavaScript	<code>js()</code>	Compiles your Kotlin code into JavaScript, either for Node.js or for web browsers
^a Machines with Apple Silicon can also run these binaries under Rosetta 2, but compiling directly for Macs with Apple-designed CPUs is not supported as of this writing.		

In the next two chapters, you will use the macOS and JavaScript targets. If you want to learn more about the other targets you can use or how to configure them, check out the documentation at kotlinlang.org/docs/mpp-dsl-reference.html#targets.

Whenever you edit your Gradle build files, you need to sync the changes with IntelliJ before they can take effect.

As you did in [Chapter 20](#), click the  Gradle sync icon that appears in a floating tooltip in the top-right corner of

the editor. The error from before should be dismissed.

Now that you have declared that your code can compile for the JVM, you need to create a folder to contain any code that is specific to the Java flavor of your application. Right-click your `src` directory and select `New → Directory`. IntelliJ will offer suggestions for common folder names. Select `jvmMain/kotlin` from this menu (or type it manually and press Return).

Congratulations – your multiplatform program now compiles for the JVM. Now all that is left is to build your currency converter.

Defining Shared Code

Doubloons4Gold’s core logic will be defined in the `commonMain` directory. This will allow you to write the parts of your application that are specific to your problem – and not the platform – once, including the exchange rate, prompts shown to the user, and input requests. You can then use these implementations on all platforms without modifications. (In fact, if you accidentally tried to use a platform-specific API in this directory, IntelliJ would respond with an error.)

Get started by making a new file in `src/commonMain/kotlin` called `Converter.kt`. Create a **`convertCurrency`** function take care of your currency conversion:

Listing 24.2 Converting currencies (`commonMain/kotlin/Converter.kt`)

```
val pricePerDoubloon = Random.nextDouble(0.75, 1.5)

fun convertCurrency() {
    println("The current exchange rate is $pricePerDoubloon per doubloon")

    println("How many doubloons do you want?")
    val numberOfDoubloons = readLine()?.toDoubleOrNull()

    if (numberOfDoubloons == null) {
        println("Sorry, I don't know how many doubloons that is.")
    } else {
        val cost = pricePerDoubloon * numberOfDoubloons
        println("$numberOfDoubloons doubloons will cost you $cost")
    }
}
```

You will need to import `kotlin.random.Random`.

This code will be the core logic of your application and will be shared across every platform that Doubloons4Gold targets. But before this code can run, you need to add an entry point that calls your new **`convertCurrency`** function.

Although each of Doubloons4Gold’s entry points will call into **`convertCurrency`**, you will be defining a separate entry point for each platform you target. Creating separate entry points per application is not strictly required, but it will make your Kotlin/JVM application easier to launch from IntelliJ. It also gives you the flexibility to change how a particular platform is started, in case you need to add extra initialization logic for only one of your platforms.

Create a new file in `jvmMain/kotlin` called `Main.kt` and add a **`main`** function. Have it print a message stating that Doubloons4Gold is running inside the JVM, which will help you keep track of your targets as you define more of them, and call **`convertCurrency`**.

Listing 24.3 Defining a JVM entry point (`jvmMain/kotlin/Main.kt`)

```
fun main() {
    println("Hello from Kotlin/JVM!")
    convertCurrency()
}
```

The run icon should appear in the margin next to your **`main`** function’s definition. Run your new **`main`** function and enter a quantity to confirm that your currency converter is working. Your output should look like this:

```
Hello from Kotlin/JVM!
The current exchange rate is 1.380843418388969 per doubloon
How many doubloons do you want?
50
50.0 doubloons will cost you 69.04217091944845
```


expect and actual

So far, so good. But it would be nicer if Doubloons4Gold used currency formatting, so your output looked like this:

```
Hello from Kotlin/JVM!
The current exchange rate is $1.38 per doubloon
How many doubloons do you want?
50
50.0 doubloons will cost you $69.04
```

You want this formatting to apply across all platforms your application supports, so it belongs in your common codebase. Create a new file in `commonMain/kotlin` called `Currency.kt` and define an extension function on **Double** called **formatAsCurrency**.

As the name suggests, this function will convert a double to an appropriately formatted string for the user’s local currency. But for now, leave the implementation as a to-do. We will explain how to implement it shortly.

Listing 24.4 Formatting currencies (commonMain/kotlin/Currency.kt)

```
fun Double.formatAsCurrency(): String = TODO("Not implemented")
```

With your stubbed-out **formatAsCurrency** function in place, update **convertCurrency** to format the currency strings. (We have split the strings across two lines to fit on the printed page; you should keep your strings on one line.)

Listing 24.5 Using formatted currencies (commonMain/kotlin/Converter.kt)

```
val pricePerDoubloon = Random.nextDouble(0.75, 1.5)

fun convertCurrency() {
    println("The current exchange rate is
        ${pricePerDoubloon.formatAsCurrency()} per doubloon")

    println("How many doubloons do you want?")
    val numberOfDoubloons = readLine()?.toDoubleOrNull()

    if (numberOfDoubloons == null) {
        println("Sorry, I don't know how many doubloons that is.")
    } else {
        val cost = pricePerDoubloon * numberOfDoubloons
        println("$numberOfDoubloons doubloons will cost you
            ${cost.formatAsCurrency()}")
    }
}
```

Now, go back to the **formatAsCurrency** function and give it an implementation. In [Chapter 5](#), we mentioned that Java has a **NumberFormat** class that will do exactly what you want here: format a number based on a template, like a currency. In the other projects you have built, you had access to all of Java’s APIs, so you could have implemented your function like this:

```
fun Double.formatAsCurrency(): String =
    NumberFormat.getCurrencyInstance().format(this)
```

But while your program only targets the JVM now, your common code can only use APIs that are accessible across all the targets you will eventually compile for. Doubloons4Gold will also compile for macOS and JavaScript, and neither of them has a **NumberFormat** class that matches Java’s APIs. So your common code cannot use this Java-specific API.

On the other hand, your common code depends on the **formatAsCurrency** function, so moving its definition to your `jvmMain` source set is not an option. Implementing the formatting logic yourself is cumbersome (and ill-advised). So what are you to do?

To solve this problem, Kotlin includes two keywords for multiplatform projects: `expect` and `actual`. In your common code, you can mark functions, properties, and classes with the `expect` modifier. When you do this, you omit the implementation, much like when you define a function on an interface (as you saw in [Chapter 17](#)).

Mark **formatAsCurrency** as an `expect` function and remove its implementation.

Listing 24.6 Declaring an expect function (commonMain/kotlin/Currency.kt)

```
expect fun Double.formatAsCurrency(): String = TODO("Not implemented")
```

When you declare an `expect` function, you are telling the compiler that the implementation of that function differs on each platform. The compiler will *expect* an implementation of this function for each target you define.

IntelliJ is now warning you of an error with your **formatAsCurrency** function. The error message reads Expected function 'formatAsCurrency' has no actual declaration in module Doubloons4Gold.jvmMain for JVM. IntelliJ is alerting you that you declared an `expect` function but did not give it an implementation for all the platforms your code targets (in this case, just the JVM).

To fix this compiler error, you need to define an implementation of **formatAsCurrency** in your `jvmMain` source set. Create a new file in `jvmMain/kotlin` called `Currency.kt` and implement the conversion using Java’s **NumberFormat** class.

Listing 24.7 Declaring an actual function (jvmMain/kotlin/Currency.kt)

```
import java.text.NumberFormat

actual fun Double.formatAsCurrency(): String =
    NumberFormat.getCurrencyInstance().format(this)
```

You define your new function with the `actual` keyword. This tells the Kotlin compiler that this is the *actual* implementation of an expected function.

Everything marked as `expect` in your common code must have a matching `actual` definition for all targets. At compile time, the Kotlin compiler will combine your common sources with the sources applicable for the target you are compiling for. Using `expect` and `actual` defines an API that will be available for all targets but whose implementation is not resolved until compile time.

When you define a function, class, or property with the `actual` keyword, an  flag appears in the editor's gutter. Clicking this flag will take you to the `expect` declaration that corresponds to the actual implementation. Similarly,

an  flag appears in the gutter next to an API defined with the `expect` keyword. If you click this flag, IntelliJ displays a menu of all its actual implementations. This allows you to easily navigate to platform-specific implementations in your code.

(By the way, although the `expect` and `actual` versions of **formatAsCurrency** are both defined in files called `Currency.kt`, the filenames do not have to match exactly. However, it is conventional to use matching filenames to make your code easier to navigate. And it is required that the `expect` and `actual` definitions be placed in the same package and have the same signature so the compiler can correctly link the two definitions.)

Run `Doubloons4Gold` again. Your output will look like this:

```
Hello from Kotlin/JVM!
The current exchange rate is $1.06 per doubloon
How many doubloons do you want?
50
50.0 doubloons will cost you $53.07
```

Developers frequently use `expect` and `actual` as you have here: to create a general set of APIs that work on every targeted platform, streamlining the development process by increasing the amount of shared logic. But calling into platform code is not the only reason to use these keywords. You can also use them to define different behaviors for specific platforms, even when the implementation could have been defined in the common source set.

Your **convertCurrency** function uses the **println** and **readLine** functions to display output and gather input from the user. These familiar functions work well for your JVM code, and they will work for the Kotlin/Native version of your application as well – but they will not work for JavaScript.

In JavaScript, the **println** and **readLine** functions are both available to your Kotlin code, but they are only used as developer tools. You do not want to ask users of your website to open these tools. You will need a different way to render content on the page.

Because you will need a different behavior for JavaScript, you will need to define behaviors for each platform separately. To do this, you will again use the `expect` and `actual` keywords. Start by creating a new file in `commonMain/kotlin` called `InputOutput.kt`. In it, define two `expect` functions called **output** and **getInput**.

Listing 24.8 Expecting platform-specific behavior (commonMain/kotlin/InputOutput.kt)

```
expect fun output(message: String)

expect fun getInput(prompt: String): String
```

Next, provide matching `actual` implementations for the JVM. Create a second `InputOutput.kt` file in `jvmMain/kotlin` and implement the same two functions using **println** and **readLine**.

Listing 24.9 Java-specific behavior (jvmMain/kotlin/InputOutput.kt)

```
actual fun output(message: String) = println(message)

actual fun getInput(prompt: String): String {
    output(prompt)
    return readLine() ?: ""
}
```

With your new, platform-specific **output** and **getInput** functions in place, you can now use them in your **convertCurrency** function. Open `Converter.kt`. To quickly replace all instances of **println** with **output**, open IntelliJ's find and replace menu with Command-R (Ctrl-R). In the first field, enter `println` as the text you want to search for. In the second field, enter `output` as the replacement text. Then click `Replace All` to make this change throughout the file.

Listing 24.10 Replacing println with output (commonMain/kotlin/Converter.kt)

```
val pricePerDoubloon = Random.nextDouble(0.75, 1.5)
```

```

fun convertCurrency() {
println output("The current exchange rate is
    ${pricePerDoubloon.formatAsCurrency()} per doubloon")

println output("How many doubloons do you want?")
val numberOfDoubloons = readLine()?.toDoubleOrNull()

if (numberOfDoubloons == null) {
println output("Sorry, I don't know how many doubloons that is.")
} else {
    val cost = pricePerDoubloon * numberOfDoubloons
println output("$numberOfDoubloons doubloons will cost you
        ${cost.formatAsCurrency()}")
    }
}

```

Close the replace menu with the x at its top right, then manually replace your usage of **readLine** with **getInput**:

Listing 24.11 Using getInput (commonMain/kotlin/Converter.kt)

```

val pricePerDoubloon = Random.nextDouble(0.75, 1.5)

fun convertCurrency() {
    output("The current exchange rate is
        ${pricePerDoubloon.formatAsCurrency()} per doubloon")

    val input = output getInput("How many doubloons do you want?")
    val numberOfDoubloons = readLine()? input.toDoubleOrNull()

    if (numberOfDoubloons == null) {
        output("Sorry, I don't know how many doubloons that is.")
    } else {
        val cost = pricePerDoubloon * numberOfDoubloons
        output("$numberOfDoubloons doubloons will cost you
            ${cost.formatAsCurrency()}")
    }
}

```

Make the same change in your **main** function.

Listing 24.12 Using output (jvmMain/kotlin/Main.kt)

```

fun main() {
println output("Hello from Kotlin/JVM!")
    convertCurrency()
}

```

Run Doubloons4Gold again. Your output will not change, but you now have the flexibility to customize your program’s behavior on a platform-by-platform basis.

In this chapter, you took the first steps to build a multiplatform application. You have declared shared code and provided Java-specific implementations for the platform-dependent details of your program.

At this point, this additional setup is not providing much benefit – the same thing could have been accomplished without declaring `expect` and `actual` functions, assuming you only wanted to target the JVM. But over the course of the next two chapters, your work in this chapter will pay off as you make Doubloons4Gold support more platforms without touching the logic in your `Converter.kt` file.

With your multiplatform scaffolding in place, Doubloons4Gold is now ready to target more platforms. In this chapter, you will be adding support for macOS.

If you have been using a Mac to work through this book, you may be asking, “Have I not already been targeting macOS?” Because your applications have compiled for Java, they will run everywhere that Java runs, including on Macs with Java installed. But what about Macs that do not have Java installed?

By *natively* targeting macOS, you will build an application that can run on macOS without an additional runtime environment – Java or otherwise. When you do this, you also have full access to macOS’s built-in frameworks and APIs, which might let you accomplish things that you cannot do in Java alone.

Before you proceed, a disclaimer: Many platforms that Kotlin/Native supports impose restrictions on the host machine you are using – that is to say, the computer you are using to compile your code. In particular, if you want to target macOS, iOS, or any of Apple’s other platforms, you must use a computer running macOS. If you attempt to make the code changes in this chapter and you are not using macOS, you will run into many issues.

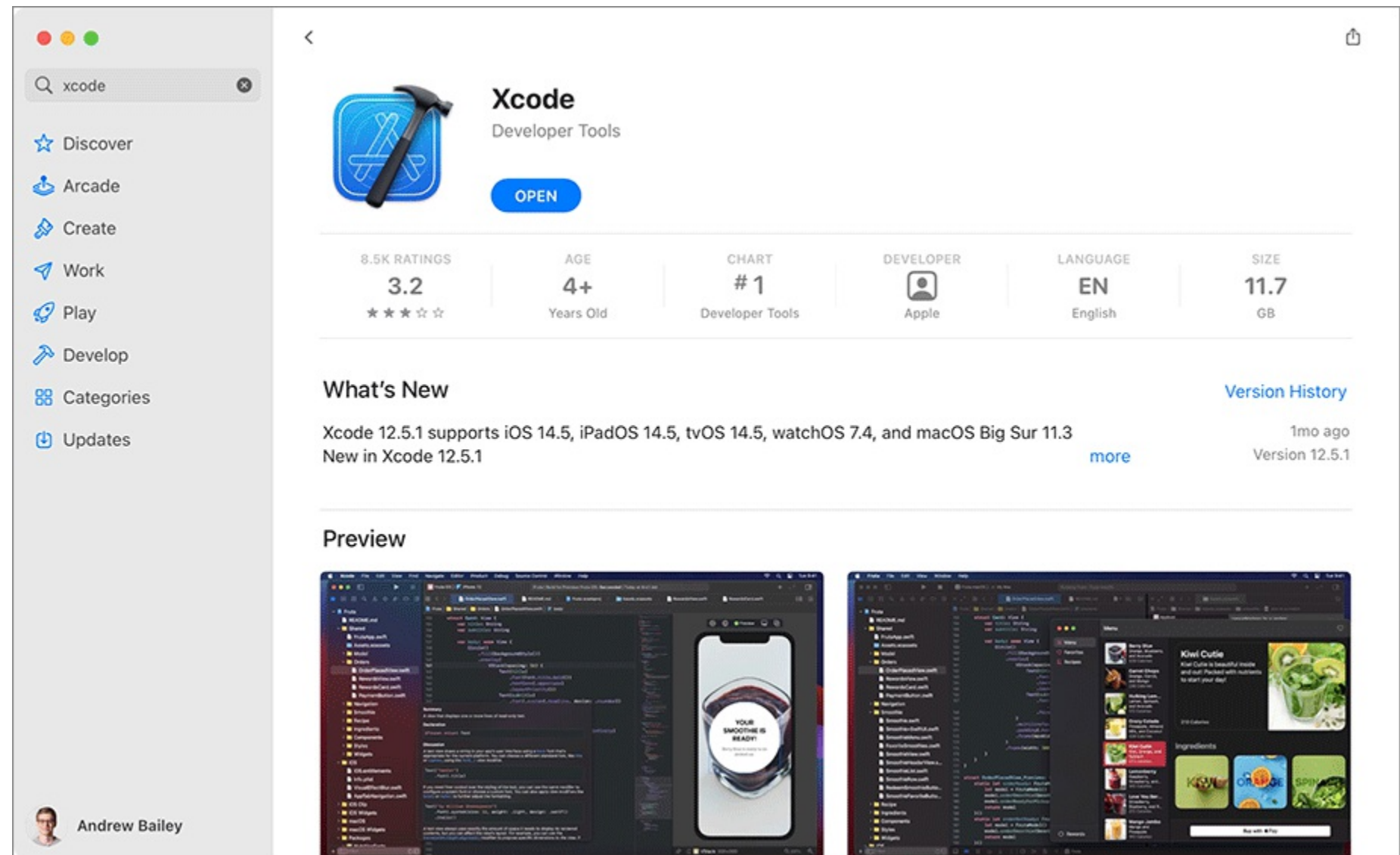
The Kotlin Multiplatform plugin disables compilation for targets that are not supported by its host, and editing in IntelliJ will be severely limited. Any classes and functions that are part of macOS’s native APIs will not be visible to IntelliJ, which causes autocompletion, quick fixes, refactoring tools, and editor inspections to not function properly – or at all.

If you do not have access to a computer running macOS, read through this chapter without making any changes to your project. The code in [Chapter 26](#) does not depend on the changes in this chapter, so the instructions there will apply to your project without adaptation.

Next, a prerequisite: The Kotlin/Native compiler requires that Xcode, Apple’s official IDE, be installed. You do not need to *use* Xcode, but it must be installed for the Kotlin/Native compiler to do its job. If it is not installed, you will get a compiler error when you attempt to build a project that targets any of Apple’s platforms.

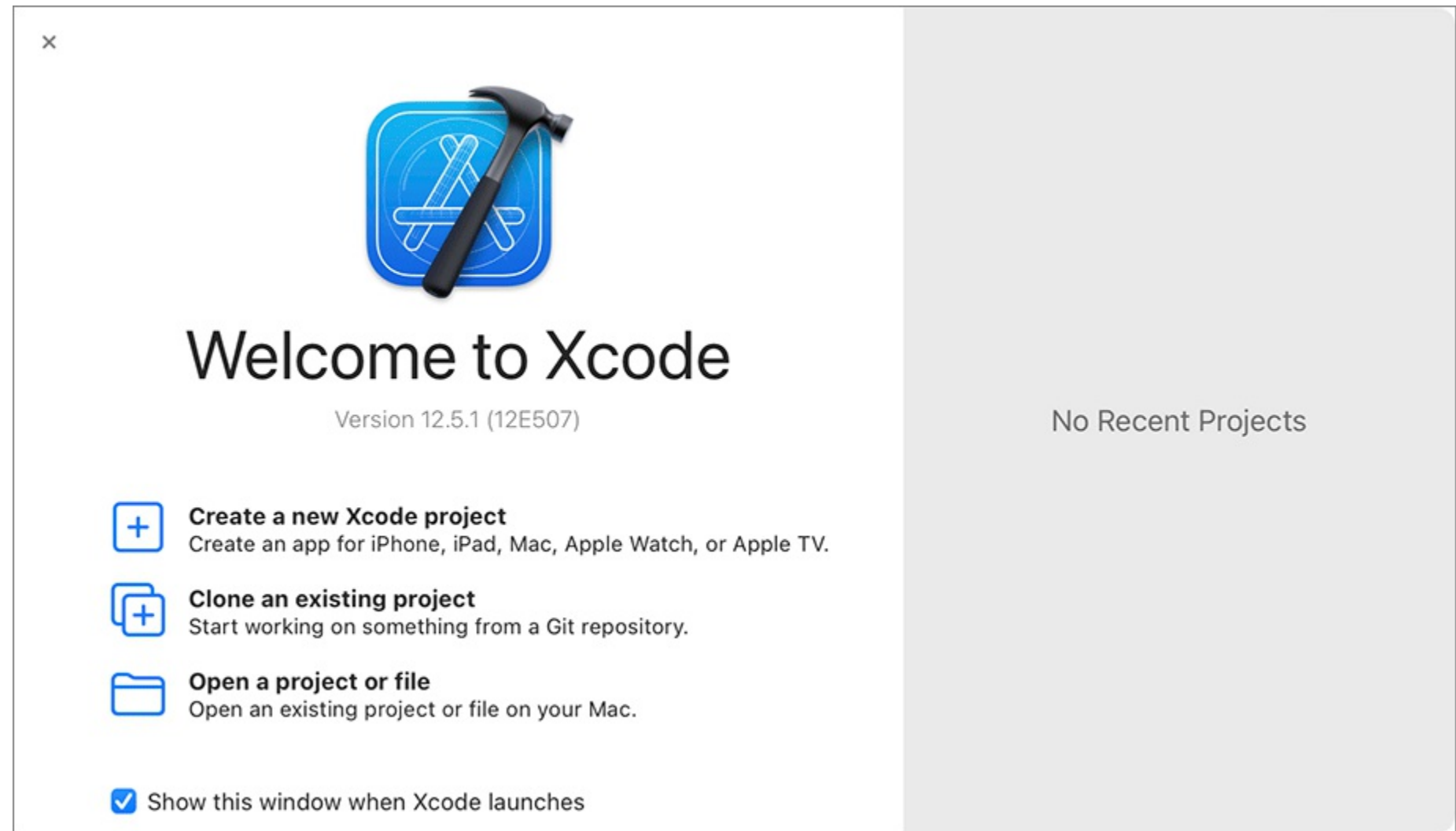
To check whether your computer has Xcode installed, open the App Store and search for Xcode. Its store listing is shown in [Figure 25.1](#). If you see an Open button, then you are good to go. If not, start downloading or updating Xcode – and be patient. Xcode is a very large application to download, and an even larger application when installed.

Figure 25.1 Xcode App Store listing



Once Xcode finishes installing, run it. You need to install the Xcode command line developer tools, which happens the first time Xcode is launched. After accepting Xcode’s license and installing these components, you will see the Welcome to Xcode screen shown in [Figure 25.2](#).

Figure 25.2 Xcode welcome screen



Once you see this screen, Xcode is ready to use, and the Kotlin/Native compiler will have everything it needs to build for macOS. Quit Xcode. You will not need it again (unless you want to take up Swift or iOS development). And, finally, a note: Kotlin’s interoperability with Apple platforms works through Objective-C, not Swift. Objective-C is an older programming language, dating back to 1984, that served as the foundation of Apple’s

software. Swift is Apple's more modern language, which has been widely adopted by the iOS and macOS community.

Although Kotlin/Native currently only supports Objective-C interoperability, Swift is also interoperable with Objective-C. This means that you *can* use Kotlin/Native and Swift together, but you might find some of the translations to be a bit awkward.

For this project, you will not be writing any Swift or Objective-C code, so this fact is not important right now. But keep this in mind if you are thinking about adopting Kotlin/Native in your Swift apps.

Declaring a macOS Target

With those preliminary issues taken care of, you are ready to add a macOS target to Doubloons4Gold. Open your `build.gradle` file and declare a new target for macOS. We will explain the additional code after you enter it.

Listing 25.1 Declaring a macOS target (`build.gradle`)

```
...
kotlin {
    jvm()
    macOSX64 {
        binaries {
            executable()
        }
    }
    sourceSets {
        ...
    }
}
```

Each platform supported by Kotlin Multiplatform offers its own options for how your project should be compiled. For Kotlin/Native targets like macOS, one of the options you can configure is the format of your binary outputs. You have several options to choose from, and you can choose more than one.

<code>executable</code>	Produces an executable binary that can be launched directly.
<code>test</code>	Produces an executable binary that can be used to run unit tests.
<code>sharedLib</code> , <code>staticLib</code>	Produce a binary for consuming your code as a library, which is helpful when you want to use your Kotlin Multiplatform code in another native project that is not written in Kotlin. Native libraries come in two flavors: shared and static. The differences between them are outside the scope of this book, but if you are already familiar with native development, you likely already know which you should use.
<code>framework</code>	Produces an Objective-C framework that can be used to consume your code as a project, likely from an Xcode project. This option is only applicable to platforms that run Objective-C, including macOS and iOS, and should usually be used instead of <code>sharedLib</code> and <code>staticLib</code> when available.

In your `build.gradle`, you specified that you want to produce an executable binary. This will let you run your program directly.

You only need to specify binaries for Kotlin/Native targets. The JVM and JavaScript do not differentiate between libraries and executables – both have the same output format per platform. This is not true for native platforms, so you provide this information to ensure that your code compiles and is consumed correctly.

With your macOS target defined, perform another Gradle sync so IntelliJ incorporates your latest changes. Once the Gradle sync completes (which can take a minute or more), make a new source directory with the path `src/macOSX64Main/kotlin`. (IntelliJ will suggest this directory when you open the New Directory menu under the `src` directory.)

Now that your code officially targets macOS and you have a home for your macOS code, you are ready to write your first code for macOS.

Writing Native Code with Kotlin

If you attempt to compile your project now, you will get a compiler error for each of your expect functions, because you have not provided actual implementations for your shiny new macOS target. You also need to define an entry point before Doubloons4Gold will function properly under macOS.

The first function you will implement is your **formatAsCurrency** function. Create a new Kotlin file called `Currency.kt`, this time in `macosX64Main/kotlin`.

In your new file, implement the same currency formatting that you set up for the JVM. You will be using some unfamiliar-looking APIs that ship with macOS. We will walk you through the code after you enter it.

Listing 25.2 Natively formatting a number (`macosX64Main/kotlin/Currency.kt`)

```
import platform.Foundation.*

actual fun Double.formatAsCurrency(): String {
    val formatter = NSNumberFormatter().apply {
        setNumberStyle(NSNumberFormatter.CurrencyStyle)
        setLocale(NSLocale.currentLocale)
    }

    val number = NSNumber(this)
    return formatter.stringFromNumber(number)!!
}
```

Your first line, `import platform.Foundation.*`, imports Apple’s Foundation framework. Foundation provides APIs across all Apple platforms for things like numbers, dates, files, data formatting, URL loading, and much more. (Foundation APIs use the `NS` prefix you see several times in this short code snippet; `NS` stands for NeXTSTEP, a 1980s ancestor of macOS.)

You then create an instance of the **NSNumberFormatter** class, which is Foundation’s equivalent to Java’s **NumberFormat** class.

You apply two options to this **NSNumberFormatter**: its number style and a locale. You set these options inside an **apply** block. This is the same **apply** function you saw in [Chapter 12](#). Recall that **apply** allows you to call functions on a receiver to configure it for use; it behaves the same here, even though it is called on a class that is not defined in Kotlin.

The number style you are setting is an enum. You access the currency style with **NSNumberFormatter.CurrencyStyle**. To set the locale, you obtain the user’s locale by accessing **NSLocale**’s `currentLocale` property and then pass it to the **setLocale** function. At this point, your **NSNumberFormatter** is fully configured.

To convert your **Double** into a formatted string, you first wrap it in an **NSNumber**, which is the base numeric type in Objective-C. Finally, you call the **stringFromNumber** function to perform the conversion.

This function can return null if the **NSNumber** is holding a value that cannot be formatted as a currency (such as if the **NSNumber** represents a Boolean). Because you are formatting a **Double**, this will never be the case, so you use the non-null assert to force the **String?** return value into the desired **String** return type.

In case you are curious, the Swift version of this extension function would look like this:

```
import Foundation

extension Double {
    func formatAsCurrency() -> String {
        let formatter = NumberFormatter()
        formatter.numberStyle = .currency
        formatter.locale = .current

        let number = NSNumber(value: self)
        return formatter.string(from: number)!
    }
}
```

And – in case you are even more curious – in Objective-C it would look like this:

```
- (NSString *)formatAsCurrency:(NSNumber *)value {
    NSNumberFormatter *formatter = [[NSNumberFormatter alloc] init];
    formatter.numberStyle = NSNumberFormatterCurrencyStyle;
    formatter.locale = NSLocale.currentLocale;
    return [formatter stringFromNumber:value];
}
```

You have two more functions that need an actual implementation: **output** and **getInput**. In [Chapter 24](#), we mentioned that **println** and **readLine** are developer tools in JavaScript, and you made **output** and **getInput** expect functions to accommodate that difference. But your macOS target can use the same implementations you use for Java.

Locate your `jvmMain/kotlin/InputOutput.kt` file in the project tool window. Select it and press Command-C (Ctrl-C) to copy it. Then select your `macosX64Main/kotlin` directory and press Command-V (Ctrl-V) to paste the file. In the dialog that appears, confirm that IntelliJ will insert a copy of the file in your `macosX64Main/kotlin` directory, as shown in [Figure 25.3](#). When you are ready, click OK to have IntelliJ make

a copy of this file.

Figure 25.3 Copying a file between source sets



IntelliJ will open the copied file in an editor. Confirm that its contents match the implementation you copied from your JVM source set:

```
actual fun output(message: String) = println(message)

actual fun getInput(prompt: String): String {
    output(prompt)
    return readLine() ?: ""
}
```

Launching a Kotlin/Native application

Now that you have defined actual implementations for each of your expected functions, your Doubloons4Gold project will compile. But you still have not defined an entry point for your macOS target, so if you attempt to execute your program, it will not know what to execute and will not start.

To fix this, you will need yet another **main** function, this time in your macOS64 source set. Create a new `Main.kt` file in `macOSX64Main/kotlin` and give it a **main** function. To differentiate your Kotlin/Native application from your Kotlin/JVM application, include an output message declaring the platform. (If you want, you can copy the `Main.kt` file from `jvmMain` into `macOSX64Main` and change the **output** string to match the following.)

Listing 25.3 Defining a main function for macOS (`macOSX64Main/kotlin/Main.kt`)

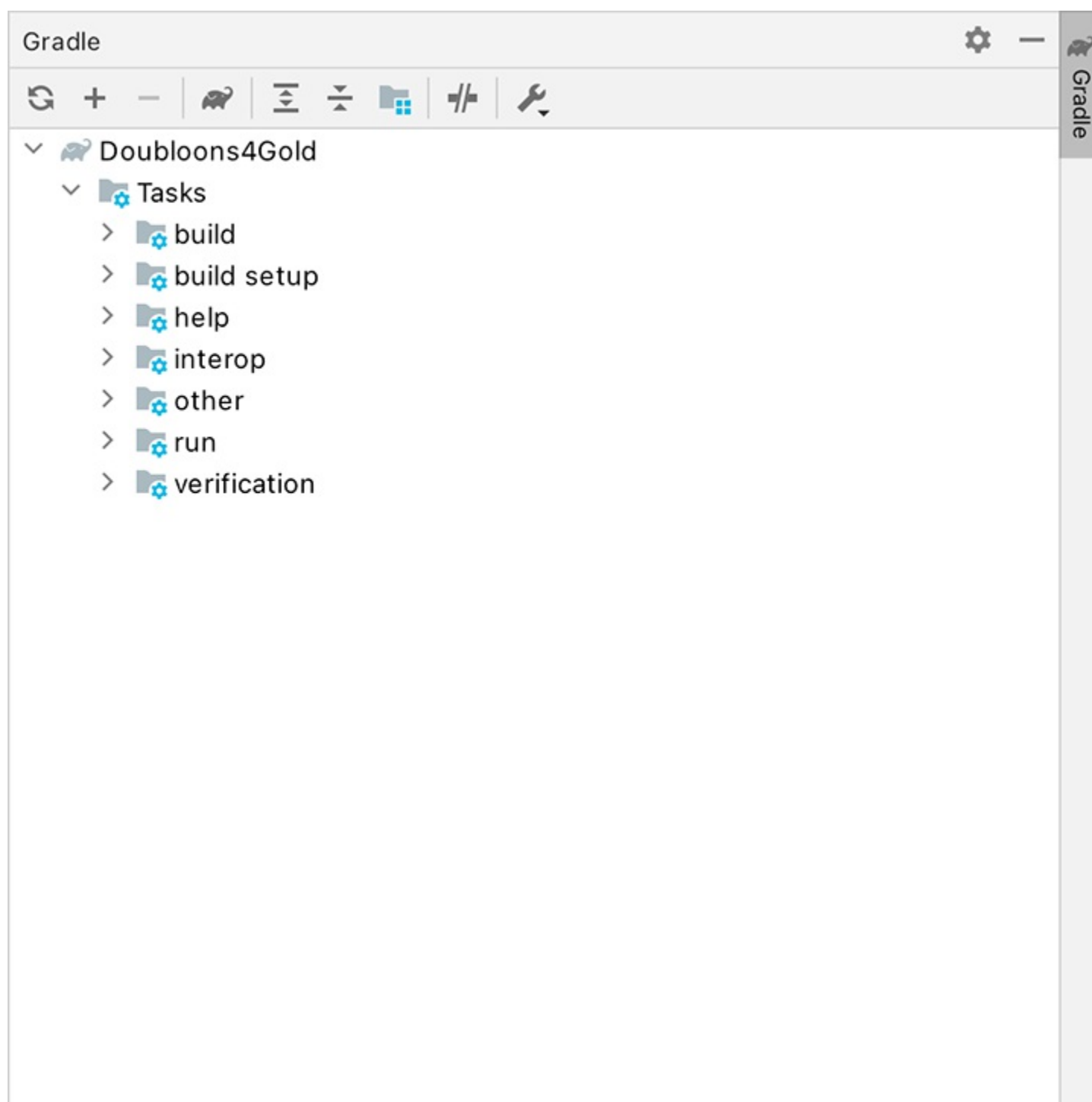
```
fun main() {  
    output("Hello from Kotlin/Native!")  
    convertCurrency()  
}
```

When you have declared new **main** functions before, IntelliJ has automatically added a run icon next to them in the margin of the editor. But look next to your new function: No run icon has appeared. IntelliJ often does not detect entry points in code that does not target the JVM, so the steps to launch your program are a bit different for your macOS target.

Instead of launching your code through IntelliJ's tooling, you will need to go directly through Gradle. Open the

Gradle tool window by clicking the  Gradle tab near the top-right corner of the IDE. You will see a panel that looks like [Figure 25.4](#).

Figure 25.4 The Gradle tool window



This tool window includes a hierarchy of all the tasks that Gradle can execute – that is to say, all the individual

build actions that Gradle is aware of for your project. These tasks are grouped into several different categories.

In the Gradle tool window, navigate to Doubloons4Gold/Tasks/run and double-click `runReleaseExecutableMacosX64` to build and run your macOS code. If you want to build your project without executing it, you can use the task called `compileKotlinMacosX64` under the build group.

You generally only need to do this once per run configuration in a project. IntelliJ will remember which run configurations you have used in the past, and you can switch between them using the dropdown next to the run button in the toolbar. After running this task, the dropdown will say `Doubloons4Gold [runReleaseExecutableMacosX64]`, indicating the run configuration that will execute when you click the run button in the toolbar.

Gradle will build and execute your macOS project. After it runs, your output should look like the following. It includes information about the Gradle build at the bottom, which we have included for reference.

```
...
> Task :runReleaseExecutableMacosX64
Hello from Kotlin/Native!
The current exchange rate is $1.14 per doubloon
How many doubloons do you want?
Sorry, I don't know how many doubloons that is.

BUILD SUCCESSFUL in 611ms
3 actionable tasks: 1 executed, 2 up-to-date
2:57:36 PM: Task execution finished 'runReleaseExecutableMacosX64'.
```

Because of the way Gradle executed your code, you might not get a chance to provide input here. That is OK. Your next step will be to locate the compiled executable on your computer and launch it manually.

Kotlin/Native Outputs

When your Kotlin code compiles for macOS, the compiler will generate a kexe (Kotlin Executable) file. This file can be executed from a terminal like any other program. Unlike the other applications you have created before, this binary can run on Macs that do not have the Java runtime environment installed. This lack of a runtime dependency reduces the complexity of your application’s installation and makes it available to a wider audience of macOS users.

After building your project, Gradle created a binary for your application in `build/bin/macosX64/releaseExecutable`. Locate this directory in the project tool window. Then right-click the `releaseExecutable` directory and select Open In → Terminal. A terminal pane will open at the bottom of IntelliJ’s window ([Figure 25.5](#)), open to the `releaseExecutable` directory you just selected.

Figure 25.5 IntelliJ’s integrated terminal



To run your application with full interactivity, enter the following command and press Return:

```
./Doubloons4Gold.kexe
```

Doubloons4Gold will start up in IntelliJ’s integrated terminal, and you will be able to properly interact with your application. The `Hello from Kotlin/Native!` message indicates that you have, in fact, created and executed a different program than your JVM version. Confirm that your program’s behavior matches the behavior you built in the JVM version by entering a value to convert:

```
> ./Doubloons4Gold.kexe
Hello from Kotlin/Native!
The current exchange rate is $1.06 per doubloon
How many doubloons do you want?
20
20.0 doubloons will cost you $21.15
>
```

With that final check, your macOS application is complete. Take a moment to think about how you might have supported multiple platforms without Kotlin Multiplatform. You would have had to re-implement your **convertCurrency** function multiple times for every platform you wanted to support – possibly in a different language for each platform.

If you later wanted to change the functionality of your program, this would involve changing several separate projects, again porting your changes across different languages. Compare that to your Kotlin Multiplatform project, where you wrote your core logic once and implemented very little platform-specific logic.

Although this project has taken a bit more setup than you are used to and requires the duplication of function signatures, you can save a significant amount of time in the long run compared to individually targeting all the platforms you want to support.

In the next chapter, you will continue to expand Doubloons4Gold, adding support for JavaScript.

For the More Curious: Kotlin Multiplatform Mobile

The most common usage of Kotlin/Native is for iOS development. Because Kotlin is the official language that Android developers use, many mobile development teams are already familiar with Kotlin. A growing number of developers are choosing to use Kotlin for both their iOS and Android apps. Using Kotlin Multiplatform in this way is called *Kotlin Multiplatform Mobile*, or KMM.

The convention for using KMM is to create three areas of code: the shared code in Kotlin, an Android app (also in Kotlin), and an iOS app in Swift. This is different from the approach used by most cross-platform mobile frameworks. When using KMM, your multiplatform code should be a library containing business logic – that is, it should only implement algorithms and platform-agnostic knowledge, rather than building out entire UIs.

By using separate projects this way, you get the best of both worlds. You code mission-critical logic once and share it across platforms, ensuring that both applications behave consistently. If there is a bug in this logic, you can fix it on both platforms simultaneously.

You also still have access to the native tooling for the platforms you are targeting. That means you get the benefits of using libraries like SwiftUI and Jetpack Compose, both of which can only be used when developing natively in the platform’s official language.

However, there are downsides when using KMM – more often affecting iOS developers than Android developers, who have the benefit of being able to work on both Android features *and* common features using their familiar language and IDE. iOS developers, on the other hand, will need to learn Kotlin in addition to Swift. They will also need to switch between Xcode and IntelliJ when a feature requires working in both the Kotlin shared library and the iOS workspace.

Also, the tooling for Kotlin Multiplatform and Kotlin/Native are still maturing and do have some rough edges. In particular, interoperability between Kotlin and Swift requires a translation layer that does not always expose Kotlin code in the most idiomatic way for Swift consumers.

For example, suspending functions are exposed to Swift code with callbacks, making their usage more complex outside of Kotlin. If you have adopted the coroutines, channels, or flows APIs, you will likely need to provide wrappers to make your asynchronous code easier to use in iOS.

Caveats aside, KMM can be a powerful tool when used carefully – especially if your team is stronger with Android development than iOS development. If the idea of code sharing interests you, we encourage you to give KMM a try. But we do recommend starting small, because it is easy to quickly introduce unnecessary complexity into your app, which might outweigh the benefits you get from sharing code.

For the More Curious: Other Native Platforms

When writing Kotlin/Native code for macOS and iOS, you have the luxury of using APIs from Objective-C. These APIs are much higher level than the APIs you would see if you were calling into code that was written in a lower-level language like C or C++. But Kotlin does support calling into lower-level languages, if you need to.

When working with low-level native code, you will almost immediately need to call APIs that require you to use tools like pointers and manual memory allocation. These topics go outside the scope of the book, but they are available in Kotlin. There are also many nuanced rules about how Kotlin and C interoperability work, including how types and functions are mapped back and forth between languages. If this interests you, take a look at the `cinterop` tool.

Windows is an example of a C-based platform with APIs at a lower level than those you saw in this chapter. In case you are *really* curious, we have included the Windows implementation of **`formatAsCurrency`** below. This code snippet demonstrates many of the C interoperability features at your disposal.

It is perfectly OK if these concepts are unfamiliar. We have included them to give you a starting point for what tools to research, but mastering the C or C++ language requires a book of its own.

```
import kotlinx.cinterop.*
import platform.windows.GetCurrencyFormatEx
import platform.windows.LOCALE_NAME_USER_DEFAULT
import platform.windows.WCHARVar

actual fun Double.formatAsCurrency(): String {
    // Convert the input to an unformatted string to use with GetCurrencyFormatEx
    val numberToFormat = toString()

    return memScoped {
        // Resolve the user's selected locale
        val userLocaleName = LOCALE_NAME_USER_DEFAULT
            ?.reinterpret<UShortVar>()
            ?.toString()

        // Figure out how many characters will appear in the formatted output
        val length = GetCurrencyFormatEx(userLocaleName, 0u, numberToFormat,
            null, null, 0)

        // Allocate space for the formatted output
        val output = allocArray<WCHARVar>(length)

        // Format the input as currency, saving the result in the output array
        GetCurrencyFormatEx(userLocaleName, 0u, numberToFormat, null, output,
            length)

        // Convert the output from a char array to a Kotlin string and return it
        output.toString()
    }
}
```


Your last task in Doubloons4Gold will be to add one final target: the web. Supporting web browsers allows your code to run virtually everywhere. There are several ways of running code on the web; you will use Kotlin/JS to compile your Kotlin code into JavaScript files that will run in client browsers.

Because your code will be running in a browser, Doubloons4Gold's interface will change a bit. Relying on **readLine** and **println** is not an option, because those functions require interacting with developer tools in JavaScript. Instead, you will be building out a basic UI inside a web page.

Declaring Support for Kotlin/JS

Your first step is to declare a new target in your code for JavaScript. This will once again involve updating your `build.gradle` file.

Listing 26.1 Declaring a JavaScript target (`build.gradle`)

```
...
kotlin {
    jvm()
    macosX64() {
        binaries {
            executable()
        }
    }
    js {
        browser()
        binaries {
            executable()
        }
    }
    sourceSets {
        ...
    }
}
...
```

The only thing new here is the **browser** function call. This specifies that you intend to run your JavaScript inside a browser. You can also use **nodejs** to have your project compile for Node.js, a standalone JavaScript runtime environment commonly used to build web servers.

Perform a Gradle sync after making these changes.

Once again, your project will no longer compile, because your expect functions do not have actual implementations in your JavaScript source set. Your code for JavaScript needs a directory of its own, just like your code for macOS and JVM. Right-click the `src` directory, select `New → Directory`, and double-click `jsMain/kotlin`.

Doubloons4Gold's behavior will be quite different when running in a web page. Instead of relying on console output, you will dynamically insert content into your web page as your code executes. You will also prompt users with in-browser alerts to get input, instead of relying on console input.

To break this problem down into smaller steps, you will begin by stubbing out all your expect functions so your program will compile. For the time being, these stubs will redirect your output to the developer console and will provide a value to simulate the user entering input. You will revisit these implementations later in the chapter.

Start by making a new Kotlin file called `InputOutput.kt` in `jsMain/kotlin`. Provide actual implementations for **output** and **getInput**:

Listing 26.2 Stubbing out I/O (`jsMain/kotlin/InputOutput.kt`)

```
actual fun output(message: String) {
    println(message)
}

actual fun getInput(prompt: String): String {
    val input = "10.0"
    return input
}
```

Next, create a new Kotlin file called `Currency.kt` in `jsMain/kotlin` and stub out your **formatAsCurrency** function. For now, return the double as a string with no additional formatting.

Listing 26.3 Stubbing out currency formatting (`jsMain/kotlin/Currency.kt`)

```
actual fun Double.formatAsCurrency(): String {
    return toString()
}
```

With your actual functions in place, your project will compile. But you need an entry point to run it – this time for code that will execute when the `Doubloons4Gold.js` script is loaded by the browser. Create a Kotlin file called `Main.kt` in `jsMain/kotlin` and use it to call **convertCurrency**.

Listing 26.4 Defining a JavaScript entry point (`jsMain/kotlin/Main.kt`)

```
fun main() {
    output("Hello from Kotlin/JS!")
    convertCurrency()
}
```

Like the other **main** functions you have written, this one will execute when your page loads.

There is one more task you need to complete before you can run your Kotlin/JS app. JavaScript cannot be executed by a browser in a vacuum – it must be hosted in a web page. You need to create this page for your code to do anything useful.

Create a new directory in `jsMain` called `resources`. (You can right-click `src` and select `New → Directory` to be offered a list of common directories, as you have done before, or you can right-click `jsMain` to create the directory

by entering its name manually.) A `resources` directory is used for files your program needs that are not executable code.

In `resources`, create an `index.html` file. (Do not use IntelliJ's HTML file template; select `New` → `File` and enter the full name, including the `.html` extension.) This file will be the default page that a web browser will open when launching `Doubloons4Gold`. Give it some simple HTML contents:

Listing 26.5 Creating a landing page (`jsMain/resources/index.html`)

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Doubloons4Gold</title>
  </head>
  <body>
    <h1>Doubloons4Gold</h1>
    <script src="Doubloons4Gold.js"></script>
  </body>
</html>
```

This HTML markup defines a very basic landing page. In the `<head>`, you set the browser-visible page title to `Doubloons4Gold`. In the `<body>`, you define a header with the same application name. You also use the `<script>` tag to import and execute your application.

Because your `<script>` tag is at the end of the `<body>`, it will load and run after the HTML that defines your page has finished loading. This helps ensure that the browser has enough time to set up the page before attempting to execute any code that might rely on it. (Other resources like images might still be loading, but this usually does not affect the execution of your code, and it will not matter for `Doubloons4Gold`.)

The `Doubloons4Gold.js` script will be generated by the Kotlin/JS compiler when you build your project. The name of this file is determined by the name of your project, but it is possible to change it if you want to. (For details, look up `outputFileName` in Kotlin's documentation at kotlinlang.org/docs/home.html.)

The Kotlin/JS flavor of `Doubloons4Gold` can now be run. To launch your code, you will need to go through Gradle, as you did when building and running your code for macOS.

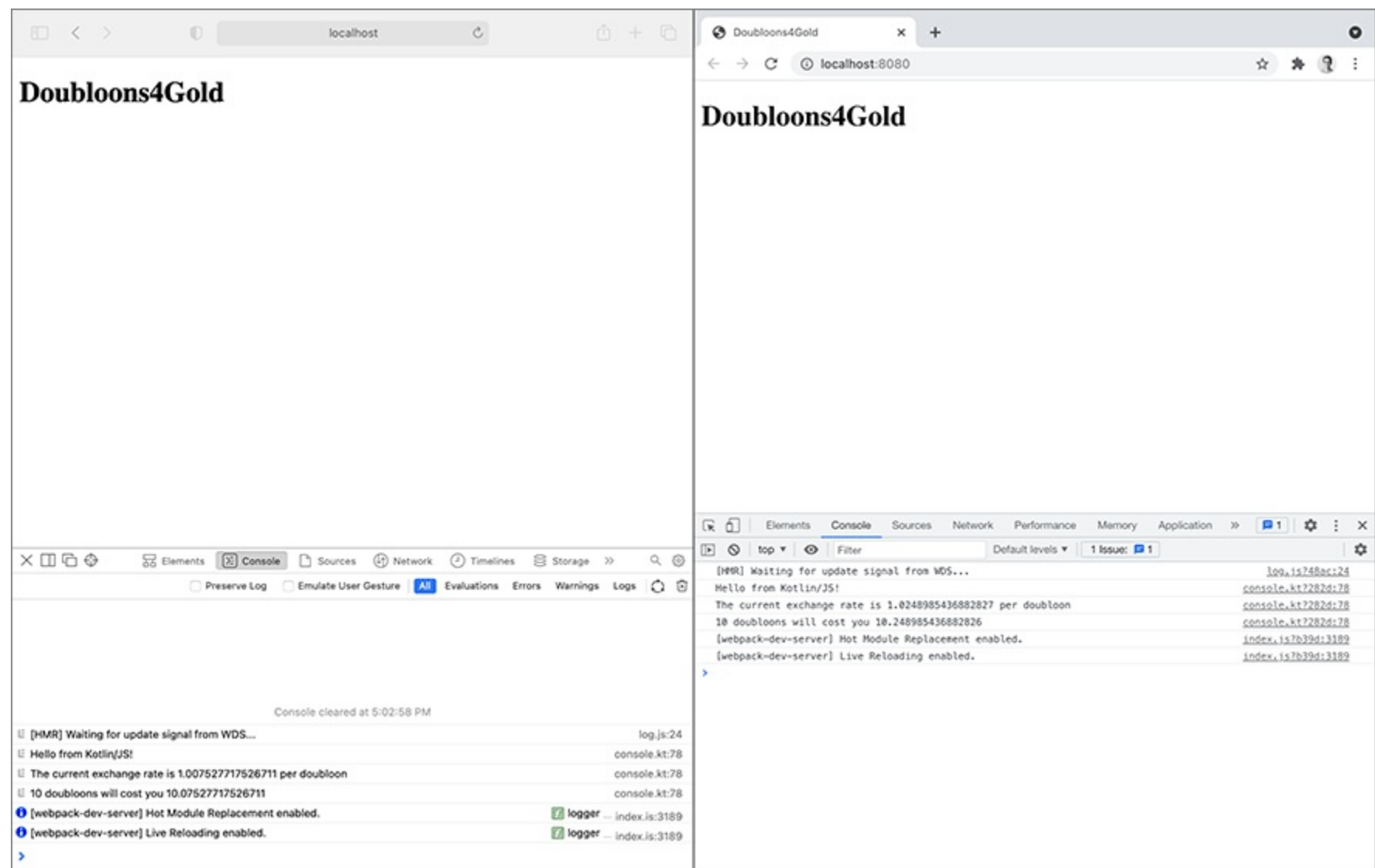
Open the Gradle tool window to view the build tasks for your project. Click the disclosure arrow to expand the other category, then double-click the `jsRun` task. Gradle will build your program and launch a web browser pointing at your live `index.html`.

When your browser launches, you will see a new tab with the title `Doubloons4Gold`. You will also see a heading with the same name, and the rest of the page will be blank. Your output will log to the developer console in IntelliJ.

To see the output in your browser, you need to *inspect* the web page. In most browsers, you can right-click anywhere on the page and select a menu item labeled `Inspect` or `Inspect Element`. For Safari users, you will first need to enable developer tools.

To do so, open `Safari` → `Preferences...` and click the `Advanced` tab. Finally, check `Show Develop menu` in menu bar and close the preferences box. Now you can right-click in the browser window and select `Inspect Element`. The developer tools will appear ([Figure 26.1](#)).

Figure 26.1 Developer tools in Chrome and Safari



With the inspector open, navigate to the Console tab. The console output should include the familiar messages about the exchange rate and final cost, as shown below. (If the console is blank, you may need to refresh the page to collect output.)

```
Hello from Kotlin/JS!           console.kt:78:16
The current exchange rate is 1.377781434913309 per doubloon  console.kt:78:16
10 doubloons will cost you 13.777814349133092               console.kt:78:16
```

This is the output from **convertCurrency**, which demonstrates that Doubloons4Gold is, in fact, running on the web for the first time. You may see other messages in this log area related to the build system used by Gradle to launch the web server. You can ignore these unrelated messages.

Running Doubloons4Gold starts a web server on your computer, which will not stop until it is told – even if the browser tab is closed. If you leave it running, it may try to reload the web page after every change to your code, which will quickly become annoying and may interfere with some of your work. Each time you run

Doubloons4Gold and return to IntelliJ, stop the program manually by clicking the  stop button in IntelliJ’s toolbar.

Interacting with the DOM

Now that Doubloons4Gold is running in the browser, your next task is to revisit the functions you stubbed out before and provide a better user experience – one that does not involve opening the developer tools. You will start by updating your `InputOutput.kt` implementations to render content on the page. Then you will address the currency formatting in **`formatAsCurrency`**.

In your **`output`** function, you need to update the page to show the line of text being displayed. To do this, you will be interacting with the DOM, a set of APIs that let you programmatically modify pages declared in HTML. To make this task easier, you will use the Kotlinx HTML library, which offers a Kotlin-friendly API for declaring HTML elements in your Kotlin code.

With Doubloons4Gold stopped, declare a dependency for this library in the `jsMain` block of your `build.gradle` file. (Ongoing builds can interfere with IntelliJ’s ability to sync with Gradle.)

Listing 26.6 Adding Kotlinx HTML library dependency (`build.gradle`)

```
...
kotlin {
    jvm()
    macosX64() {
        binaries {
            executable()
        }
    }
    js {
        browser()
        binaries {
            executable()
        }
    }
    sourceSets {
        commonMain {

        }
        commonTest {

        }
        jsMain {
            dependencies {
                implementation "org.jetbrains.kotlinx:kotlinx-html-js:0.7.3"
            }
        }
    }
}
```

This dependency declaration is a bit different than what you have seen before. Because you only need to manipulate the DOM in your Kotlin/JS application, you declared this dependency for the `jsMain` source set. It will only apply to that build output. This cuts down on the final output size of your other applications, since they will not use this dependency. If you add a dependency to your `commonMain` or `commonTest` blocks, it is available regardless of the platform you target.

Sync with Gradle to make IntelliJ aware of your new dependency.

Now, update your **`output`** and **`getInput`** functions to add content to your page. Open `jsMain/kotlin/InputOutput.kt`.

To get a hold of the DOM and make edits to the page, you use the `document` property. This property is declared in JavaScript and contains many functions and properties that affect how your page renders. To access the `<body>` content of your HTML from your Kotlin code, you use the `body` property on `document`.

To add new content, you will use the **`append`** function. **`append`** is a function defined in JavaScript for the `body` property, but you will instead use an overload from the Kotlinx HTML library you just added to your project. This overload uses a lambda expression to define the content you are adding, and it is much more expressive and idiomatic than the built-in JavaScript APIs.

Update **`output`** and **`getInput`** to append formatted text to the page. We will explain the syntax after you enter it.

Listing 26.7 Appending paragraphs (`jsMain/kotlin/InputOutput.kt`)

```
import kotlinx.browser.document
import kotlinx.html.dom.append
import kotlinx.html.*

actual fun output(message: String) {
    val body = checkNotNull(document.body) {
        "Could not locate the <body> tag"
    }

    body.append {
        p { +message }
    }

    println(message)
}

actual fun getInput(prompt: String): String {
    val body = checkNotNull(document.body) {
        "Could not locate the <body> tag"
    }

    val input = "10.0"

    body.append {
        p {
            em { +prompt }
        }
    }
}
```

```

        +" "
        strong { +input }
    }
}
return input
}

```

Inside these **body.append** blocks, you call several functions (also from the Kotlinx HTML library) to add and manipulate content. To add a paragraph, you call the **p** function; to make text italic, you call **em** (for “emphasize”), and to make text bold you call **strong**. These function names correspond to the names of the HTML tags you are adding to the page (<p>, , and).

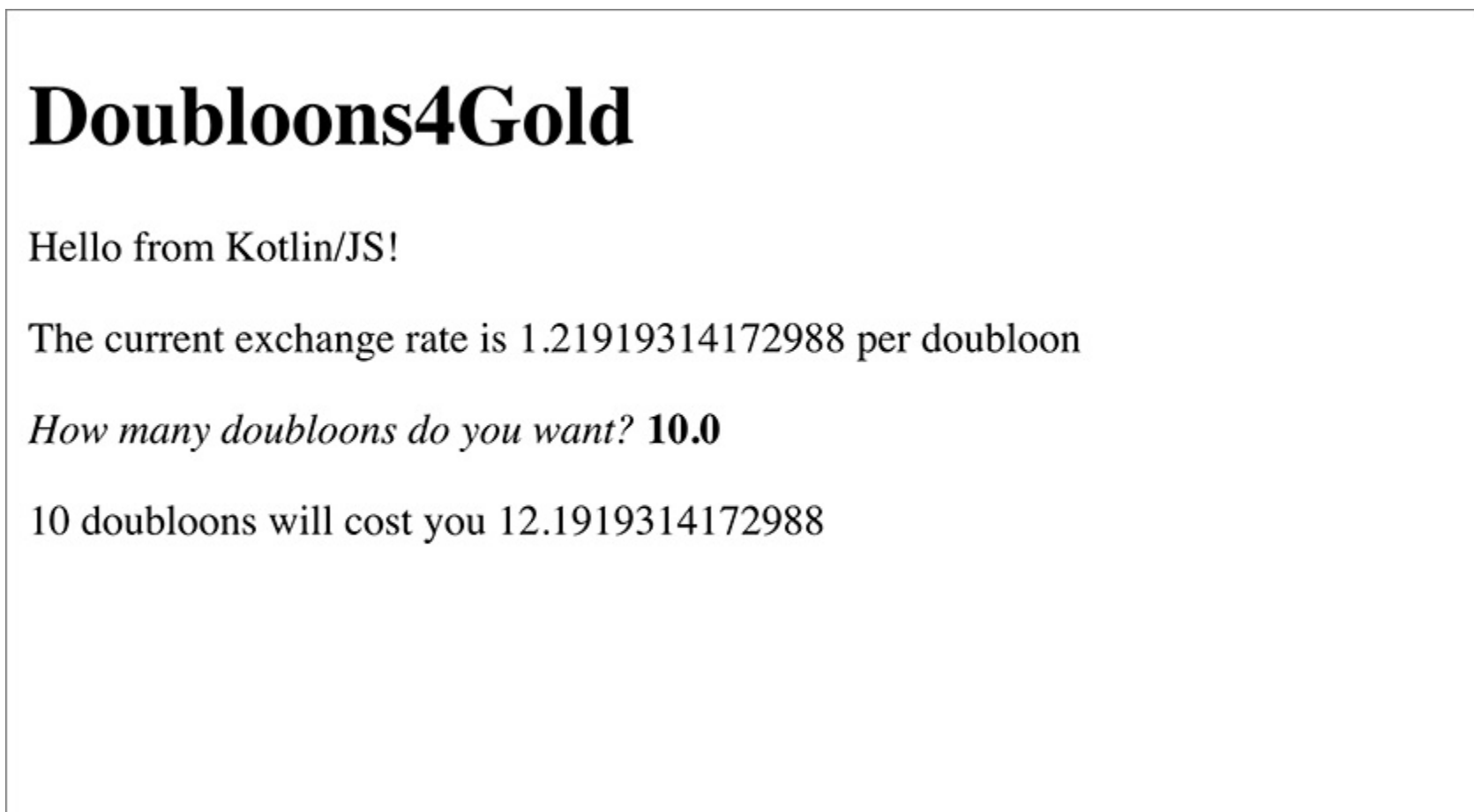
To specify that a string should be inserted into an element as text, you prefix it with the unary plus (+) operator.

Putting it all together, **output** now adds a paragraph to the **body** with the contents of the **message** passed to it, and **getInput** adds a paragraph to the **body** with the passed-in **prompt** text in italics followed by a space and the **input** in bold.

Re-run Doubloons4Gold.

Your web page will now update dynamically. It should look something like this:

Figure 26.2 A dynamically updated page



Now that you have modified the DOM so output appears on the page, you can prompt users for input. To do this, you will use another variable in the DOM, **window**.

window has a **prompt** function that shows an alert (a pop-up window) to the user with an input box; it returns the text that is entered. That is what you will use to collect input from the user about how many doubloons they want.

To show the user the exchange rate and the final cost, you will use **window**’s **alert** function. This will display an alert to the user that does not accept any input; it has an OK button to dismiss it.

Insert calls to **prompt** and **alert** to complete your **output** and **getInput** implementations.

Listing 26.8 Alerting users (jsMain/kotlin/InputOutput.kt)

```

import kotlinx.browser.document
import kotlinx.browser.window
import kotlinx.html.dom.append
import kotlinx.html.*

actual fun output(message: String) {
    val body = checkNotNull(document.body) {
        "Could not locate the<body> tag"
    }

    body.append {
        p { +message }
    }

    println(message)
    window.alert(message)
}

actual fun getInput(prompt: String): String {
    val body = checkNotNull(document.body) {
        "Could not locate the <body> tag"
    }

    val input = "10.0" window.prompt(message = prompt, default = "") ?: ""

    body.append {
        p {
            em { +prompt }
            +" "

```

```
        strong { +input }
    }
    return input
}
```

Run Doubloons4Gold again. Once the page loads, you will get an alert with the message The current exchange rate is *[rate]* per doubloon. When you click OK, you will get another alert with the message How many doubloons do you want? and a text field.

Enter any number, then click OK. You will see one more alert with the message *[Number]* doubloons will cost you *[cost]*. When you click OK, the page will update with a record of the exchange.

(Because of how the **alert** function and the DOM are implemented, your page does not re-render between messages. It renders after **convertCurrency** function returns. You could update your code to give the page an opportunity to redraw itself after showing an alert, but this would make your code much more complicated.)

Your **output** and **getInput** functions are now complete. But Doubloons4Gold still needs a bit of work. When your web page prints out a monetary amount, it prints it as a decimal number with no formatting or rounding. Your next task will enable Doubloons4Gold to call into JavaScript's internationalization APIs to perform this formatting.

The external Keyword

JavaScript has a number of internationalization APIs under the **Intl** object. You will be using the **Intl.NumberFormat** class, which can format currency for a given locale. You can find its documentation at developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/NumberFormat.

Try using this class in your `Currency.kt` file. Do not attempt to add any import statements. You will see errors after you make this change, which we will explain afterward.

Listing 26.9 Using an unknown JavaScript class (`jsMain/kotlin/Currency.kt`)

```
actual fun Double.formatAsCurrency(): String {  
    return toString()  
    val numberFormatter = Intl.NumberFormat()  
  
    return numberFormatter.format(this)  
}
```

Kotlin/JVM and Kotlin/Native both have interoperability features that automatically allow you to call APIs declared by the platform and by external libraries while still enjoying all the type safety and analysis tools that Kotlin and IntelliJ offer. Although Kotlin/JS also has interoperability features that allow you to call JavaScript code from Kotlin (and vice versa), you cannot simply call JavaScript and still get all those Kotlin and IDE benefits the way you can with Kotlin/JVM and Kotlin/Native.

To access any class or function declared in JavaScript and still get all the benefits of the Kotlin compiler and the IDE, the JavaScript API must be included in the Kotlin/JS standard library, be defined in a Kotlin library that you are including, or have its stubs manually declared in your code. **Intl.NumberFormat** does not meet any of these criteria, which is why you cannot access it.

In cases like this, Kotlin is not even aware of the type's existence – so using the API is not as simple as the import statements you used for Kotlin/JVM and Kotlin/Native code.

To address this, you need to make Kotlin aware of the **Intl** class using the `external` keyword. Create a new Kotlin file called `Intl.kt` in `jsMain/kotlin` to hold the class and function headers for **Intl.NumberFormat**. These headers inform Kotlin of the APIs that are available from JavaScript.

Listing 26.10 Declaring an external class (`jsMain/kotlin/Intl.kt`)

```
external class Intl {  
    class NumberFormat {  
        constructor()  
  
        fun format(number: Number): String  
    }  
}
```

When you mark a class, function, or property as `external`, you indicate to Kotlin that it is declared *externally*, meaning that it is declared and implemented in JavaScript. External classes and functions cannot have implementations in your Kotlin code. They exist exclusively to inform the compiler about APIs that are defined in JavaScript. With the definition you just provided, you now have access to **Intl.NumberFormat**'s constructors and **format** function.

(The `external` keyword [or `extern`, in some languages] is not unique to Kotlin/JS, but Kotlin/JS has a specific meaning for this keyword. The `external` keyword is *only* used to declare APIs that were not defined in Kotlin so they can be called from Kotlin code. In addition to the usage you just saw, you can also use the `external` keyword on the JVM when using the Java Native Interface [JNI], which is outside the scope of this book.)

When you use the `external` keyword, the compiler will take your word on the types and functions you are declaring. It is up to you to ensure that these definitions do not have typos and exactly match the definitions in JavaScript, otherwise your program will encounter runtime errors because you will be trying to access APIs that do not exist. The benefit of the `external` keyword is that once you (carefully) define these declarations, the compiler can then ensure you are not making typos from that point forward. For more information on JavaScript's built-in types, see Mozilla's developer documentation at developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.

There are other APIs on both **Intl** and **Intl.NumberFormat** that are available in JavaScript but not included in your external definition. That is OK. You only need to declare functions and classes that you intend to call from your Kotlin code. Any functions or classes that are not included in an external definition remain unknown to the compiler and cannot be called directly from your Kotlin code.

Run `Doublings4Gold` and confirm that your project now builds without errors. Your output will now use **Intl.NumberFormat**'s default formatting for numeric values and look something like this:

```
The current exchange rate is 0.845 per doubloon
```



```
How many doubloons do you want? 1.5
1.5 doubloons will cost you 1.268
```

Your output looks much friendlier now that it is rounding to three decimal points. This is a nice win, but you want your **Intl.NumberFormat** to be configured to output currency string. This will require a bit more effort and a few more JavaScript interoperability features.

Executing Raw JavaScript

To specify that you want your numbers to be formatted as currency, you will need to use a different constructor than the one you stubbed out in the external definition of **Intl.NumberFormat**. This new constructor will take two arguments: a locale and an optional set of formatting specifications. To begin, specify the **locale** parameter. You will declare the **options** parameter later (for now, your code will use the default set of options).

Listing 26.11 Declaring a second constructor (jsMain/kotlin/Intl.kt)

```
external class Intl {  
    class NumberFormat {  
        constructor()  
  
        constructor(locale: String)  
  
        fun format(number: Number): String  
    }  
}
```

This parameter specifies the locale of the user. It affects formatting options like the thousands and decimal separators. You can obtain the locale of the user’s browser with the `navigator.language` property.

Much like **Intl.NumberFormat**, the `navigator` property is not known by Kotlin out of the box. One option to access this property is to declare additional external stubs to inform Kotlin of these APIs. This will work, but can be cumbersome for a one-off call to a more basic API.

As an alternative to defining the external interfaces in Kotlin, you can also evaluate arbitrary JavaScript in your Kotlin code. You do this by calling the **js** function. This function takes in a single **String** argument containing the JavaScript code you wish to evaluate. Take it for a spin by introducing a new property to `Currency.kt` called `userLocale`.

Listing 26.12 Executing raw JavaScript (jsMain/kotlin/Currency.kt)

```
private val userLocale: String  
    get() = js("navigator.language") as String? ?: "en-US"  
  
actual fun Double.formatAsCurrency(): String {  
    val numberFormatter = Intl.NumberFormat(  
        locale = userLocale  
    )  
  
    return numberFormatter.format(this)  
}
```

Your `userLocale` property will invoke the `navigator.language` code and return its value. This property is null when the language is unknown. If this happens, you will fall back to US English.

Inlining raw JavaScript like this comes with several disadvantages. For one, the Kotlin compiler is not able to validate any JavaScript code defined this way. It is up to you to ensure that you have not made any typos, otherwise you will not find out until your program executes.

Kotlin is also not able to infer the return type of expressions evaluated using the **js** function. Here, you explicitly cast the result to a **String?**. As the developer, you must ensure that this type information matches what will happen when your program runs, otherwise you will get type errors and unexpected behavior.

We recommend you use the **js** function sparingly. That said, there are scenarios where it makes sense to inline raw JS code – typically in one-off situations like this one where the effort required to define the external stubs does not match the value you gain in having these APIs. If you needed to use the `navigator` or `language` property somewhere else in your code, it might be better to define external classes instead of relying on **js**.

Run `Doubloons4Gold` again and confirm that your output has not changed.

Dynamic Types

Way back in [Chapter 2](#), we mentioned that Kotlin uses a static type checking system. This means that every variable, function parameter, return type, and expression in Kotlin has a type that is known at compile time. This information is either inferred by the compiler or declared explicitly in your Kotlin code.

The static type checking system allows the compiler to confirm that you are accessing behaviors that exist for the kind of data you are working with – a powerful ability. Many of IntelliJ’s code editing, analysis, and refactoring tools build on the static type system.

By contrast, JavaScript’s type system is *dynamic*. In a dynamic type system, types are validated at runtime instead of compile time. Also, JavaScript does not have any built-in syntax to declare a variable’s type or a function’s return type.

There are several benefits to using a dynamic type system. Your code will have fewer errors that appear before it runs, which can allow for faster iteration times. In some cases, dynamic types can also speed up certain code changes or refactoring, since you can swap out your types with just a few edits to your code.

But dynamic types are a double-edged sword. Without the safety of the compiler checking that you are using types correctly, it is easy to make changes that inadvertently break parts of your code. And these errors will only appear at runtime, meaning that testing your code becomes even more critical when making changes to a return value or variable type.

When using Kotlin/JS, the language remains statically typed, but you also have the option of using dynamic types in your Kotlin code. For an example, let’s revisit the **format** function. In JavaScript, you might call this function like this:

```
const price = 25.68;
const locale = navigator.language ?? "en-US";
const formatOptions = { style: 'currency', currency: 'USD' };
const numberFormat = new Intl.NumberFormat(locale, formatOptions);

numberFormat.format(price);
```

This syntax might look a little unfamiliar, but everything in this snippet has a direct Kotlin equivalent that you have already seen. Take a look at the `formatOptions` declaration: The righthand side of this assignment creates a new object with two properties, `style` and `currency`. Because JavaScript uses a dynamic type system, the **Intl.NumberFormat** class can read these two properties even though this object does not extend from a particular base class or interface.

Think back to your new constructor. You already provided a `locale` parameter, and you now need an `options` parameter. `options` does just what it says: It defines optional criteria that affect how a value should be formatted. It is where you will declare that you want to format the value as a currency. But what type should it be?

Because this parameter is used dynamically, there are no specifications about what types can be used – any value could be passed in as the `options`, regardless of whether it is actually a useful input. For scenarios like this, you can use a special type only available to Kotlin/JS called **dynamic**. Add an additional parameter to your constructor, using this new type.

Listing 26.13 Declaring a dynamic parameter (jsMain/kotlin/Intl.kt)

```
external class Intl {
    class NumberFormat {
        constructor()

        constructor(locale: String, options: dynamic)

        fun format(number: Number): String
    }
}
```

You can use **dynamic** as a type anywhere in your Kotlin/JS code, though we recommend only using it as an interoperability tool for calling code written in JavaScript. (By the way, the return type of the **js** function is **dynamic**, which was why you had to explicitly cast its result to a **String?**.)

We mentioned earlier that the `options` parameter is optional. In fact, JavaScript, much like Kotlin, has the notion of default parameters. To expose default parameters in your Kotlin code, you can use the value `definedExternally` to indicate that a value exists, but is defined externally. Update your **Intl.NumberFormat** constructor to use a default value for the `options` parameter.

Listing 26.14 Declaring a default external parameter (jsMain/kotlin/Intl.kt)

```
external class Intl {
    class NumberFormat {
        constructor()

        constructor(locale: String, options: dynamic = definedExternally)

        fun format(number: Number): String
    }
}
```

```
}
```

`definedExternally` is often used when an external function has default arguments, but it can also be used for variables that are defined in JavaScript. If you want, you can also declare `definedExternally` as the body for an external function, as long as it is the only statement in the body.

With the `options` parameter in place, you can now provide an argument for it. There are several ways to go about creating the `options` object in your Kotlin code. One way is to use the `js` function again. If you wanted to go this route, the function call would look like this:

```
val options: dynamic = js("{ style: 'currency', currency: 'USD' }")
```

But to demonstrate the power of dynamic types in Kotlin, you will instead define this object in Kotlin and use it dynamically. You can use an object expression to declare the parameters in Kotlin.

Do so now, specifying the currency style and US dollar as the currency. (JavaScript does not have an API to determine the user’s preferred currency. If this is critical for your application, you will need to ask the user to select a currency or infer it based on the user’s country of residence.)

Listing 26.15 Using an object dynamically (`jsMain/kotlin/Currency.kt`)

```
private val userLocale: String
    get() = js("navigator.language") as String? ?: "en-US"

actual fun Double.formatAsCurrency(): String {
    val numberFormatter = Intl.NumberFormat(
        locale = userLocale,
        options = object {
            @JsName("style") val style = "currency"
            @JsName("currency") val currency = "USD"
        }.asDynamic()
    )
    return numberFormatter.format(this)
}
```

You included `@JsName` annotations on your properties. When Kotlin compiles your code into a JavaScript file, it can minify the output code to reduce the file size of the generated script. A consequence of this is that variable names are mangled after compilation. This is an issue, because the `Intl.NumberFormat` class will resolve properties by their names, which must match after compilation. By annotating properties with `@JsName`, you inform the compiler of the names you want the properties to have after your code is compiled.

You also called `asDynamic` to treat your object as a dynamic type. This is optional, since the `options` parameter’s type is already `dynamic`, but helps to make your code more explicit about when a value will be used dynamically. When you call `asDynamic` on a value, it is converted to the `dynamic` type.

You can still call functions and read properties on `dynamic` values, but be warned: IntelliJ and the Kotlin compiler will not perform any validation to ensure that the functions and properties you access exist. And the Kotlin compiler will not attempt to resolve any extension functions or properties for a dynamic type. This also applies to scope functions, so if you attempt to call a function like `let` or `apply` on a dynamic type, the compiler assumes that the function exists on the type.

Run `Doubloons4Gold` again. Your output should now look like this:

```
The current exchange rate is $1.42 per doubloon
How many doubloons do you want? 25
25 doubloons will cost you $35.44
```

`Doubloons4Gold` is complete. With Madrigal able to get a hold of doubloons, she can now kick off her vacation and enjoy the island’s splendors.

Reflect on what you have accomplished in the past three chapters. You wrote a single application that runs in three different places: the JVM, macOS, and the web. Each of these variations uses platform-specific APIs to provide an appropriate experience for the user. And you accomplished all this without rewriting your mission-critical `convertCurrency` function.

For the More Curious: Front-End Frameworks

There are many popular frameworks for building a web UI, with some of the most popular being React, Vue, and Angular. You can technically use any of these libraries with the interoperability features you learned about in this chapter. JetBrains also provides Kotlin/JS bindings for React, which saves you the trouble of building this compatibility layer yourself.

However, we do not recommend using these frameworks from Kotlin directly. Using a front-end JavaScript framework in Kotlin often leads to more complex code than the equivalent JavaScript code. There are also substantially more resources and guides for these frameworks in plain-old JavaScript than there are for Kotlin.

Remember, Kotlin Multiplatform is most powerful when you can share critical logic between applications. Instead of writing a web UI in Kotlin, we recommend using Kotlin/JS to build a library that you can consume from another JavaScript application. To learn more about how to call Kotlin code from JavaScript, take a look at the documentation at kotlinlang.org/docs/js-to-kotlin-interop.html.

Challenge: Currency Exchange Fees

Right now, Doubloons4Gold exchanges currencies at the current exchange rate. It does not allow any markup in the price, which is not a profitable strategy for a currency exchange business.

Implement a new policy where a 5% exchange fee is charged on each currency exchange. As a second challenge, set a minimum currency exchange fee of \$5 to encourage customers to use fewer transactions. After this change, your output should look something like this.

```
Hello from Kotlin/JS!
The current exchange rate is $1.19 per doubloon
How many doubloons do you want?
5
5.0 doubloons is worth $5.96
It will cost $10.96 for 5.0 doubloons

Hello from Kotlin/JS!
The current exchange rate is $1.37 per doubloon
How many doubloons do you want?
100
100.0 doubloons is worth $137.27
It will cost $144.13 for 100.0 doubloons
```

Make sure this policy affects all customers, regardless of whether they are using the web, JVM, or macOS application. You do not want any loopholes or special treatment for customers based on which platform they use.

Afterword

That is it. You have learned the fundamentals of the Kotlin programming language. Pat yourself on the back!
This is where the real work begins.

Where to Go from Here

Kotlin is a language that can be used in many contexts, be it as a replacement for your back-end server code, the language driving your hot new Android app, or a way to share code between applications on different platforms. At this point, you likely have an idea of where you will use your new knowledge, so *use it*. That is the key to making the most of this book and writing good Kotlin code.

If you are looking for Kotlin documentation to dig into, we recommend kotlinlang.org. For reference material, we hold *Kotlin in Action* (manning.com/books/kotlin-in-action) in high regard.

You do not have to write code alone: Kotlin's community is vibrant and excited about the future of the language. Kotlin is open source, so if you would like to see it developed in real time (or even contribute), you can find it on GitHub: github.com/jetbrains/kotlin. We encourage you to reach out to local Kotlin user groups or, if your community does not have one, start one.

Shameless Plugs

If you would like to follow up with the authors, you can find us on Twitter. Andrew is @_andrewbailey, David is @drgreenhalgh, and Josh is @mutexkid.

If you would like to know more about Big Nerd Ranch, take a look at bignerdranch.com. We offer a bevy of other great guides, which you can find at bignerdranch.com/books. Might we suggest *Android Programming: The Big Nerd Ranch Guide*? Android development is a great way to put your newfound Kotlin knowledge to use.

We also offer intensive training courses and develop apps for clients. If you can dream up a way to use some great code, Big Nerd Ranch can help.

Thank You

Last, we just have to say thank you. Without you – yes, *you* – this book would not be possible.

We hope that you have enjoyed reading it as much as we have enjoyed writing it. Now go out there and write the next great application in Kotlin.

Glossary

accumulator variable

A temporary variable used to calculate an operation on a series of values.

addition and assignment operator

Adds or appends the value on its righthand side to the variable on its lefthand side: +=

algebraic data type

A type that allows the representation of a closed set of possible subtypes, such as an enumerated class.

(See also class, enumerated; class, sealed)

application entry point

The starting place for a program. In Kotlin, this is the **main** function.

argument

An input to a function.

argument, default

A value assigned to a function argument to be used if no value is provided by the caller.

argument, named

A function argument passed with the name of the parameter it is being sent as.

arrow operator

An operator used in lambda expressions to separate parameters from the function body, in when expressions to separate the condition from the branch, and in function type definitions to separate parameter types from return types: ->

assignment operator

Assigns the value on its righthand side to the variable on its lefthand side: =

branch

A set of code executed conditionally.

bytecode

The lower-level language used by the Java Virtual Machine.

called on, implicitly

Called on a receiver that is scoped but not specified.

(See also scoping, relative)

casting

See type casting, safe cast operator, smart casting.

channel

A mechanism for transferring values between coroutines using suspending **send** and **receive** functions.

channel, buffered

A channel with a buffer of specified size.

channel, conflated

A channel with a buffer size of one that replaces the existing value if a new value is sent.

channel, rendezvous

A channel with no buffer.

channel, unlimited

A channel with a buffer whose size has no specified upper bound.

class

A definition of a category of objects represented in code.

class body

The portion of a class definition, designated by curly braces, that holds its behavior and data definitions.

class function

A function defined within a class.

class property

An attribute on a class representing the state or characteristics of an object.

class, abstract

A class that cannot be instantiated directly but is used to define common functionality among its subclasses.

class, data

A class with special features for data management.

class, enumerated

A class defining a collection of constants called enumerated values; all instances of the class are of one of the defined values.

(See also class, sealed; value, enumerated)

class, nested

A named class defined within another class.

class, sealed

A class with a defined set of subtypes, allowing the compiler to check whether a when expression contains an exhaustive set of branches.

(See also algebraic data type; class, enumerated)

class, value

A class that wraps a single value to provide additional functionality and a different interpretation of the original type.

code comment

A note in code; comments are ignored by the compiler.

cold flow

A flow that does no work until it starts to be collected from.

(See also flow, hot flow)

collection, eager

A collection whose values are accessible when it is instantiated.

(See also collection, lazy)

collection, lazy

A collection whose values are generated only as needed.

(See also collection, eager; function, iterator)

comparison operator

Compares the elements on its lefthand and righthand sides.

compilation

The translation of source code into a lower-level language to create an executable program.

compile time

See compilation.

compile-time error

An error that occurs during compilation.

(See also compilation)

compiled language

A language that is translated into machine-language instructions by a compiler prior to executing.

(See also compilation; compiler)

compiler

A program that performs compilation.

(See also compilation)

conditional expression

A conditional statement assigned to a value that can be used later.

console

A pane in the IntelliJ IDEA window that displays information about what happened when a program was executed, along with any outputs from the program.

(See also run tool window)

constant

A val that holds a value that cannot be changed after the program is compiled.

constructor

A special function that prepares a class for use during instantiation.

constructor, primary

A class constructor defined in the class header; any other constructors for the class must delegate to the primary constructor.

consumer

A class with a generic type that is inputted to the class but not outputted by it.

contravariance

A generic type that appears as a consumed type.

(See also consumer)

control flow

Rules for when code should be executed.

coroutine

A Kotlin feature that allows work to be performed in parallel.

coroutine builder

A function that creates a coroutine.

(See also coroutine)

coroutine scope

An instance of **CoroutineScope** that controls the execution of a coroutine.

covariance

A generic type that appears as a produced type.

(See also producer)

decrement operator

Decrements the value of a variable by 1: --

delegate

A way of defining a property using a template for its behavior.

destructuring

Declaring and assigning multiple variables from a single value with a single expression.

dot syntax

Syntax that connects two elements with a dot (.); used when calling a function defined on a type and when referring to a class property.

dynamic type system

A type system in which types are validated at runtime versus compile time.

editor

The main area of the IntelliJ IDEA window, where code can be entered and edited.

encapsulation

The principle that an object's functions and properties should be visible to other objects only as needed. Also the process of hiding function and property implementations using visibility modifiers.

equality, referential

Of two variables: referring to the same type instance.

(See also equality, structural)

equality, structural

Of two variables: having the same value.

(See also equality, referential)

escape character

Distinguishes characters that have special meaning to the compiler: \

escape sequence

A sequence of characters that is escaped so that it has a meaning other than the literal characters it contains.

(See also escape character)

exception

A disruption to the execution of a program; an error.

exception, unchecked

An exception that can be thrown without requiring a function to try/catch it or declare that it throws the exception.

exhaustive

Including all possibilities.

expression

A combination of values, operators, and functions that produces another value.

extend

Gain functionality through inheritance or interface implementation.

extension

A property or function declared on a type without inheritance.

external

Code that is declared and implemented outside Kotlin.

field

Storage for the data associated with a property.

floating point number

A number represented using a decimal that can be positioned at an arbitrary place based on its significant digits.

flow

A stream of data that can be subscribed to.

format string

A string with placeholders for data. These placeholders can specify formatting rules used when converting the inputs to text.

function

A reusable portion of code that accomplishes a specific task.

function body

The portion of a function definition, designated by curly braces, that holds its behavior definitions and return type.

function call

A line of code that triggers a function and passes it any necessary arguments.

function call, chainable

A function call that returns its receiver or another object that a subsequent function can be called on.

function header

The part of a function definition that includes the visibility modifier, function declaration keyword, name, parameters, and return type.

function inlining

A compiler optimization commonly used to reduce the memory overhead for functions that accept anonymous functions as arguments.

function literal

See lambda expression.

function overloading

Defining two or more function implementations with the same name and scope but a different set of parameters.

function reference

A named function converted to a value that can be used as a function literal.

function type

The type of a function literal, defined by its input, output, and parameters.

function, abstract

A function declared without an implementation in an abstract class.

(See also class, abstract)

function, aggregation

A function that reduces the contents of a collection into a single value.

function, anonymous

A function defined without the `fun` keyword; often used as an argument to another function.

(See also function, named; lambda)

function, combining

A function that takes multiple collections and combines them into a single new collection.

function, composable

A function that can be combined with other functions.

function, extension

A function that adds functionality to a particular type.

function, filter

A function that works on the contents of a collection by applying a predicate function to check a condition for each element; elements for which the predicate returns true are added to a new collection returned by the filter function.

function, higher-order

A function that takes another function as an argument or returns a function as a result.

function, infix

A function that can be called without a dot or the parentheses around its argument.

function, iterator

A function that allows a series of data to be traversed in order.

function, mutator

A function that changes the contents of a mutable collection.

function, named

A function defined with a name and the `fun` keyword.

(See also [function, anonymous](#))

function, precondition

A Kotlin standard library function that defines conditions to check before executing the following code.

function, single-expression

A function with a single expression in its body.

(See also [expression](#))

function, transform

In functional programming, a function that works on the contents of a collection by transforming each element using its transformer function; transform functions return a modified copy of the collection they are called on.

(See also [functional programming](#))

function, transformer

In functional programming, the anonymous function passed to a transform function that specifies the action to be taken on each element in the collection the transform is called on.

(See also [functional programming](#))

functional programming

A style of programming that relies on higher-order functions, often on collections, that can be combined to create complex behavior.

generic type

A type used by a class or function that is specified at the call site.

generic type parameter

The parameter specified for a generic type, such as `<T>`.

generics

A type system feature that allows functions and types to work with many different types.

getter

A function defining how a property is read.

hot flow

A flow that is always active, regardless of whether it is being collected from.

(See also [flow, cold flow](#))

imperative programming

A programming paradigm in which sequences of executable statements are executed in order.

increment operator

Adds 1 to the value of the element it is affixed to: `++`

index

An integer corresponding to the position of an element in a series.

indexed access (get) operator

Gets the element at a particular index from a collection: `[]`

inheritance

An object-oriented programming principle in which the properties and functions of classes are shared by their subclasses.

initialization

Preparation of a variable, property, or class instance for use.

initialization, late

Initialization of a variable that is delayed until its value is assigned.

initialization, lazy

Initialization of a variable that is delayed until it is first accessed.

initializer block

A block of code, prefixed with `init`, that will be executed during initialization of an object instance.

instance

A particular occurrence of an object.

instantiate

Create an instance of.

interface

A set of abstract functions and properties used to create common features among objects not related by a superclass.

interoperate

Interact with another programming language natively.

iteration

Repeating a process, as for each element in a range or collection.

Kotlin REPL

A tool in IntelliJ IDEA that allows code to be tested without creating a file or running a complete program.

Kotlin standard library functions

A set of functions built into the Kotlin programming language.

lambda

Another term for an anonymous function.

(See also `function`, `anonymous`)

lambda expression

Another term for an anonymous function's definition.

(See also `function`, `anonymous`)

lambda result

Another term for an anonymous function's return.

(See also `function`, `anonymous`)

logical operator

A function or operator symbol that performs a logical operation on its input(s).

logical 'and' operator

Returns true if and only if the elements on its lefthand and righthand sides are both true: `&&`

logical 'not' operator

Returns true if the element it is appended to is false and false if the element it is appended to is true: `!`

logical 'or' operator

Returns true if either of the elements on its lefthand and righthand sides is true: `||`

method

Java terminology for a function.

(See also function)

module

A discrete unit of functionality that can be run, tested, and debugged independently.

modulus operator

Returns the remainder when one number is divided by another; also called the remainder operator: %

mutable

Able to be changed.

(See also read-only)

mutual exclusion

The practice of blocking off an area of code so only one thread can access it.

non-null assertion operator

Calls a function on a nullable element, returning an exception if the element it is called on is null: !!

non-nullable

Unable to be assigned a null value.

null

Nonexistent.

null coalescing operator

Returns the element on its lefthand side if it is non-null; otherwise returns the element on its righthand side: ?:

nullable

Able to be assigned a null value.

object declaration

A named singleton created with the object keyword.

(See also object, companion; object expression; singleton)

object expression

An unnamed singleton created with the object keyword.

(See also object, companion; object declaration; singleton)

object, companion

An object defined within a class and marked with the companion modifier; companion objects allow their members to be accessed by referencing the outer class name only.

(See also object declaration; object expression; singleton)

operator (for flows)

A function that changes how a flow emits items.

operator (symbolic)

A symbol that represents a mathematical, comparison, or logical operation.

operator overloading

Defining an implementation for an operator function on a custom type.

override

Provide a custom implementation for an inherited function or property.

parameter

An input required by a function.

parameterized type

The type specified by the call site to use for a generic type.

pass an argument

Provide an input to a function.

platform type

Ambiguous types returned to Kotlin from Java code; they may be nullable or non-nullable.

polymorphism

The ability to use the same named entity (such as a function) to produce different results.

predicate

A true/false condition provided to a function as a lambda to define how work should be performed.

producer

A class with a generic type that is outputted by a class but not inputted to it.

project

All the source code for a program, along with information about dependencies and configurations.

project tool window

The pane on the left of the IntelliJ IDEA window that shows a project's structure and files.

property, computed

A property defined such that its value is computed each time it is accessed.

property, inline

A class property defined in the primary constructor.

race condition

A condition that causes unexpected application behavior due to the same state being modified simultaneously by two or more elements in a program.

range

A sequential series of values or characters.

range to operator

Defines an inclusive range between two numbers or characters: ..

(See also range)

read-only

Able to be read but not written.

(See also mutable)

receiver

The subject of an extension function.

receiver type

The type an extension adds functionality to.

refactor

Change the presentation or location of code without changing its functionality.

referential equality operator

Evaluates whether the variable on its lefthand side points to the same type instance as the value on its righthand side: ===

(See also equality, referential)

regular expression, regex

A defined character search pattern.

remainder operator

See modulus operator.

reserved keyword

A word that cannot be used as a name in code.

return type

The type of data a function outputs after completing its work.

return, implicit

Data that is returned without an explicit return statement.

run tool window

A pane in the IntelliJ IDEA window that displays information about what happened when a program was executed along with any outputs from the program.

(See also console)

runtime

When a program is executed.

runtime error

An error that occurs after compilation, during program execution.

safe call operator

Calls a function only if the element it is called on is non-null: `?.`

safe cast operator

Attempts to cast one type as another and returns null if the cast is invalid: `as?`

(See also type casting)

scope

The portion of a program in which an entity, such as a variable, can be referred to by name.

scoping, relative

The scoping of standard function calls within a lambda to the receiver the lambda is called on.

(See also called on, implicitly)

setter

A function defining how a property's value is assigned.

signed numeric type

A numeric type that includes both positive and negative values.

singleton

An object declared with the `object` keyword; singletons are limited to a single instance throughout program execution.

smart casting

The tracking by the compiler of information that has been checked for a branch of code, such as whether a variable has a null value.

(See also type casting)

spread operator

Causes the elements of an **Array** to be treated as individual parameters to functions that accept a variable number of arguments: `*`

statement

An instruction in code.

string

A sequence of characters.

string concatenation

Combining two or more strings in a single output.

string interpolation

Using a string template.

string template

Syntax that allows a variable name to stand in for its value in a string.

string, raw

Syntax that allows all characters and whitespace in a string literal to be represented as entered.

structural equality operator

Evaluates whether the value on its lefthand side is equal to the value on its righthand side: ==

(See also equality, structural)

structured concurrency

A paradigm for managing batches of coroutines and handling them collectively as resources.

(See also coroutine)

subclass

A class defined as inheriting properties from another class.

subtraction and assignment operator

Subtracts the value on its righthand side from the variable on its lefthand side: -=

superclass

The class that a subclass inherits from.

target (a platform)

Design a program to run on a platform.

thread

A low-level component that executes an application's instructions in the order they are written.

(See also coroutine)

thread pool

A collection of threads available to be assigned work.

(See also coroutine, thread)

thread, main

The thread that controls work presented to the user; also called the UI thread.

throw (an exception)

Generate an exception.

type

A classification of data; a variable's type determines the nature of the values it can hold.

type casting

Treating an object as though it were an instance of a different type.

(See also safe cast operator, smart casting)

type checking

Confirmation by the compiler that the value assigned to a variable is of the correct type.

type checking, static

Type checking performed as code is edited or compiled versus at runtime.

type erasure

The loss of type information for generics at runtime.

type inference

The ability of the compiler to recognize a variable's type based on the value assigned to it.

type system, static

A system in which the compiler labels source code with type information for checking.

type, collection

A data type that represents a group of data elements, such as a list.

Unicode character

A character defined in the Unicode standard.

value, enumerated

A possible value for an instance of an enumerated class.

(See also class, enumerated)

variable

An element that holds a value; variables may be read-only or mutable.

variable, file-level

A variable defined outside any function or class.

variable, local

A variable defined within a function's scope.

visibility

The accessibility of an element from other code elements.

visibility modifier

A modifier added to function and property declarations to set their visibility.

zero-indexed

Using the value 0 for the first index (in a series or collection).

Index

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Z](#)

Symbols

!

- for platform types, [Interoperability and Nullity](#)
- logical ‘not’ operator, [Logical operators](#)

!! (double-bang/non-null assertion operator), [Option three: the non-null assertion operator](#)

!= (non-equality operator), [if/else Statements](#)

!== (referential non-equality operator), [if/else Statements](#)

""", for raw strings, [Raw Strings](#)

\$, for string interpolation/templating, [String Interpolation](#)

% (modulus/remainder operator), [Integers](#)

&& (logical ‘and’ operator), [Logical operators](#)

() (function invocation operator), [Operator Overloading](#)

*

- multiplication operator, [Operator Overloading](#)
- spread operator, [zip](#)

+

- addition operator, [Operator Overloading](#)
- for string concatenation, [String Interpolation](#)

++ (increment operator), [Declaring a Variable](#), [Operator Overloading](#)

+= (addition and assignment (plus assign) operator)

- about, [Declaring a Variable](#)
- overloading, [Operator Overloading](#)
- with lists, [Changing a list’s contents](#)
- with maps, [Adding Entries to a Map](#)
- with sets, [Adding elements to a set](#)

- (subtraction operator), [Operator Overloading](#)

-- (decrement operator), [Declaring a Variable](#), [Operator Overloading](#)

-= (subtraction and assignment (minus assign) operator)

- about, [Declaring a Variable](#)
- overloading, [Operator Overloading](#)
- with lists, [Changing a list’s contents](#)
- with maps, [Adding Entries to a Map](#)
- with sets, [Adding elements to a set](#)

-> (arrow operator)

- in function type definitions, [The function type](#)
- in lambda expressions, [Function arguments](#)
- in when expressions, [when Expressions](#)

. (dot), for class property references, [Class Properties](#)

.. (range to operator), [Ranges](#), [Operator Overloading](#)

/ (division operator), [Operator Overloading](#)

/*...*/, for code comments, [Lists and Sets](#)

//, for code comments, [Logical operators](#)

: operator

- for interface implementation, [Implementing an Interface](#)
- for subclassing, [Creating a Subclass](#)

:: operator, for function references, [For the More Curious: Function References](#)

< (less-than operator), [if/else Statements](#)

<= (less-than-or-equal-to operator), [if/else Statements](#)

<>, for parameterized type definitions, [Lists](#)

= (assignment operator)

- for maps, [Adding Entries to a Map](#)
- for variable values, [Declaring a Variable](#)

in single-expression function syntax, [Single-Expression Functions](#)
== (structural equality operator), [if/else Statements](#), [Operator Overloading](#)
=== (referential equality operator), [if/else Statements](#)
> (greater-than operator), [if/else Statements](#), [Operator Overloading](#)
>= (greater-than-or-equal-to operator), [if/else Statements](#)
?. (safe call operator), [Option two: the safe call operator](#)
?: (null coalescing operator), [The null coalescing operator](#)
@, for labels, [For the More Curious: Return Labels](#)
@file:JvmName annotation, [Beyond Classes](#)
@JvmField annotation, [Beyond Classes](#)
@JvmOverloads annotation, [Beyond Classes](#)
@JvmStatic annotation, [Beyond Classes](#)
@NotNull annotation, [Interoperability and Nullity](#)
@Nullable annotation, [Interoperability and Nullity](#)
@Throws annotation, [Exceptions and Interoperability](#)
[] (indexed access (get) operator)
 overloading, [Operator Overloading](#)
 with lists, [Accessing a list's elements](#)
 with maps, [Accessing Map Values](#)
[]= (set operator), [Changing a list's contents](#)
\\ (escape character), [String Interpolation](#)
_, for private backing properties, [MutableStateFlow](#)
| (pipe), in raw strings, [Raw Strings](#)
|| (logical 'or' operator), [Logical operators](#)

A

abstract classes

about, [Interfaces and Abstract Classes](#), [Abstract Classes](#)
abstract functions, [Abstract Classes](#)
interface implementation, [Abstract Classes](#)
subclassing, [Abstract Classes](#)
vs interfaces, [Abstract Classes](#)

abstract functions, [Abstract Classes](#)

abstract keyword, [Abstract Classes](#)

actual keyword, [expect and actual](#)

add function, [Changing a list's contents](#), [Adding elements to a set](#)

addAll function, [Changing a list's contents](#), [Adding elements to a set](#)

addition and assignment (plus assign) operator (+=)

about, [Declaring a Variable](#)
overloading, [Operator Overloading](#)
with lists, [Changing a list's contents](#)
with maps, [Adding Entries to a Map](#)
with sets, [Adding elements to a set](#)

addition operator (+), [Operator Overloading](#)

alert function, [Interacting with the DOM](#)

algebraic data types, [For the More Curious: Algebraic Data Types](#)

all function, [Lambdas and the Kotlin Standard Library](#)

also function, [also](#), [Using Scope Functions](#)

and(number) function, [For the More Curious: Bit Manipulation](#)

Android

and Kotlin/JVM, [Introduction to Kotlin Multiplatform](#)
targeting, [Formatting a Double](#), [Defining a Kotlin/JVM Target](#)

anonymous classes, [Object expressions](#)

anonymous functions (see [lambdas](#))

ANSI escape sequences, [Defining a Function That Accepts a Function](#)

Any class

about, [The Kotlin Type Hierarchy](#), [For the More Curious: Any](#)
and platform independence, [For the More Curious: Any](#)
default function implementations, [Data Classes](#)

append function, [Interacting with the DOM](#)
application entry point, [Creating your first Kotlin file](#)
apply function, [apply](#), [Using Scope Functions](#), [Writing Native Code with Kotlin](#)
arguments

(see also [functions](#))
in class constructors, default, [Default arguments](#)
in class constructors, named, [Named arguments](#)
in functions, default, [Default Arguments](#)
in functions, named, [Named Function Arguments](#)
in lambdas, [Function arguments](#), [Accepting multiple arguments](#)
vs parameters, [Calling a Function](#)

Array type, [For the More Curious: Array Types](#)

ArrayIndexOutOfBoundsException, [Index boundaries and safe index access](#)

arrow operator (->)
in function type definitions, [The function type](#)
in lambda expressions, [Function arguments](#)
in when expressions, [when Expressions](#)

as (type cast) operator, [Type casting](#)

as? (safe cast) operator, [For the More Curious: The Safe Cast Operator](#)

assert function, [Preconditions](#)

assignment operator (=)
for maps, [Adding Entries to a Map](#)
for variable values, [Declaring a Variable](#)
in single-expression function syntax, [Single-Expression Functions](#)

async function, [async and await](#), [Dividing Work with Channels](#)

atomic references, [For the More Curious: Race Conditions](#)

await function, [async and await](#), [Dividing Work with Channels](#)

B

BigDecimal type, [Floating Point Numbers](#)

bitwise operations, [For the More Curious: Bit Manipulation](#)

Boolean type, [Kotlin's Built-In Types](#)

break expression, [For the More Curious: The break Expression](#)

by keyword, [For the More Curious: Property Delegates](#)

Byte type, [Numeric Types](#)

bytecode
about, [For the More Curious: Targeting the JVM](#)
decompiling, [Inspecting Kotlin Bytecode](#)
inspecting, [Inspecting Kotlin Bytecode](#)

C

catch function, [Error Handling in Flows](#)

Channel class, [Sending to a Channel](#)

channels
about, [Channels](#)
and buffers, [Rendezvous channels](#)
and jobs, [Joining Jobs](#)
buffered, [Buffered channels](#)
closing, [Closing a Channel](#)
conflated, [Conflated channels](#)
creating, [Sending to a Channel](#)
producers and workers, [Dividing Work with Channels](#)
ReceiveChannel interface, [Receiving from a Channel](#)
receiving values from, [Receiving from a Channel](#)
rendezvous, [Rendezvous channels](#)
SendChannel interface, [Receiving from a Channel](#)
sending values to, [Sending to a Channel](#)
unlimited, [Unlimited channels](#)

vs flows, [Channels](#), [Joining Jobs](#)

Char type, [Kotlin's Built-In Types](#), [For the More Curious: Unicode](#)

check function, [Preconditions](#)

checkNotNull function, [Preconditions](#)

class keyword, [For the More Curious: Algebraic Data Types](#)

classes

about, [Classes](#)

abstract, [Abstract Classes](#)

(see also [abstract classes](#))

algebraic data types, [For the More Curious: Algebraic Data Types](#)

anonymous, [Object expressions](#) (see [object expressions](#))

Any, [For the More Curious: Any](#)

(see also [Any class](#))

Channel, [Sending to a Channel](#)

class bodies, [Class Functions](#)

class functions, [Class Functions](#)

companion objects, [Companion objects](#)

constructors, [Constructing Instances](#)

(see also [constructors](#))

CoroutineContext, [Structured Concurrency](#)

CoroutineDispatcher, [Structured Concurrency](#)

CoroutineScope, [Structured Concurrency](#)

data classes

(see also [data classes](#))

declared with object

(see also [companion objects](#), [object declarations](#), [object expressions](#))

Deferred, [async and await](#)

defining, [Defining a Class](#), [Defining the Room Class](#)

Dispatchers, [Structured Concurrency](#)

enumerated classes (enums), [Enumerated Classes](#)

extending (see [extensions](#))

instantiating, [Constructing Instances](#)

Job, [Structured Concurrency](#), [Joining Jobs](#)

Kotlin type hierarchy, [The Kotlin Type Hierarchy](#), [For the More Curious: Any](#)

MutableSharedFlow, [For the More Curious: SharedFlow](#)

MutableStateFlow, [MutableStateFlow](#)

nested, [Nested Classes](#)

object declarations, [Object declarations](#)

object expressions, [Object expressions](#)

properties (see [properties](#))

sealed classes, [For the More Curious: Algebraic Data Types](#)

SharedFlow, [For the More Curious: SharedFlow](#)

StateFlow, [MutableStateFlow](#)

subclasses, [Creating a Subclass](#)

superclasses, [Creating a Subclass](#)

type casting, [Type casting](#), [For the More Curious: The Safe Cast Operator](#)

type checking, [Type Checking](#)

value classes, [For the More Curious: Value Classes](#)

clear function, [Changing a list's contents](#), [Adding elements to a set](#)

close function, [Closing a Channel](#)

code comments

with `/*...*/`, [Lists and Sets](#)

with `//`, [Logical operators](#)

collect function, [Setting Up a Flow](#)

collection types

(see also [List type](#), [lists](#), [Map type](#), [maps](#), [Sequence type](#), [sequences](#), [Set type](#), [sets](#))

about, [Lists and Sets](#)

compared, [Iterating Through a Map](#)

creating instances, [Lists](#)

eager vs lazy, [Sequences](#)

companion modifier, [Companion objects](#)
companion objects, [Companion objects](#)
compareTo function, [Operator Overloading](#)
comparison operators
 (see also individual operator names, [operator overloading](#), [operators](#))
 about, [if/else Statements](#)
 order of operator precedence, [Logical operators](#)
compilation, [Compilation and execution of Kotlin/JVM code](#)
compile-time constants, [Compile-Time Constants](#)
compile-time errors, [Compile Time vs Runtime](#)
compiler, [Compile Time vs Runtime](#)
computed properties, [Computed properties](#)
conditional expressions
 about, [Conditional expressions](#)
 omitting curly braces in, [Removing braces from if/else expressions](#)
console, [Running your Kotlin file](#)
const modifier, [Compile-Time Constants](#)
constants
 about, [Variables, Constants, and Types](#)
 naming conventions, [Compile-Time Constants](#)
constructors
 about, [Constructors](#)
 calling, [Primary constructors](#), [Secondary constructors](#)
 default values for parameters, [Secondary constructors](#), [Default arguments](#)
 defining class properties in, [Defining properties in a primary constructor](#)
 defining initialization logic in, [Secondary constructors](#)
 for subclassing, [Creating a Subclass](#)
 named arguments in, [Named arguments](#)
 naming conventions for parameters, [Primary constructors](#)
 parameters, [Primary constructors](#)
 primary, [Primary constructors](#)
 secondary, [Secondary constructors](#)
consumer role, [in and out](#)
contains function, [Checking the contents of a list](#), [Creating a set](#), [Operator Overloading](#)
containsAll function, [Checking the contents of a list](#), [Creating a set](#)
contravariance, covariance vs in, out, [in and out](#)
control flow, [Conditionals](#)
copy function, [copy](#)
coroutines
 about, [Coroutines](#)
 and structured concurrency, [Structured Concurrency](#)
 and threads, [Coroutines](#)
 builders, [Coroutine Builders](#)
 context, [Structured Concurrency](#)
 CoroutineContext class, [Structured Concurrency](#)
 CoroutineDispatcher class, [Structured Concurrency](#)
 CoroutineScope class, [Structured Concurrency](#)
 Deferred class, [async and await](#)
 dispatchers, [Structured Concurrency](#)
 Dispatchers class, [Structured Concurrency](#)
 enabling, [Enabling Coroutines](#)
 Job class, [Structured Concurrency](#)
 Kotlinx Coroutines library, [Enabling Coroutines](#)
 Ktor library with, [Using an HTTP Client](#)
 launch function, [Coroutine Builders](#)
 modifying the context of, [Structured Concurrency](#)
 runBlocking function, [Coroutine Scopes](#)
 scope, [Structured Concurrency](#)
 suspending functions, [Structured Concurrency](#)
count function, [Anonymous Functions](#)

D

data classes

- about, [Data Classes](#)
- benefits and limitations, [Destructuring declarations](#)
- copy function, [copy](#)
- creating copies, [copy](#)
- destructuring, [Destructuring declarations](#)
- hashCode function, [equals and hashCode](#)
- implementation of library functions, [toString](#)
- toString function, [toString](#), [equals and hashCode](#)

dec function, [Operator Overloading](#)

decrement operator (--), [Declaring a Variable](#), [Operator Overloading](#)

Deferred class, [async and await](#)

delegates

- about, [For the More Curious: Property Delegates](#)
- lazy, [For the More Curious: Property Delegates](#)

destructuring, [Destructuring](#)

Dispatchers class, [Structured Concurrency](#)

distinct function, [Collection Conversion](#)

div function, [Operator Overloading](#)

division operator (/), [Operator Overloading](#)

dot syntax, for class property references, [Class Properties](#)

Double type

- about, [Kotlin's Built-In Types](#), [Numeric Types](#)
- converting from Int, [Converting Between Numeric Types](#)
- converting to Int, [Converting Between Numeric Types](#)
- formatting, [Formatting a Double](#)
- .NaN, [Floating Point Numbers](#)
- .NEGATIVE_INFINITY, [Floating Point Numbers](#)
- .POSITIVE_INFINITY, [Floating Point Numbers](#)
- precision, [Floating Point Numbers](#)
- rounding, [Floating Point Numbers](#)

double-bang/non-null assertion operator (!!), [Option three: the non-null assertion operator](#)

drop function, [Flow Termination](#)

dropWhile function, [Flow Termination](#)

E

editor, [Your First Kotlin Project](#)

elementAt function, [Creating a set](#)

emit function, [Setting Up a Flow](#)

encapsulation (see [visibility](#))

enumerated classes (enums)

- about, [Enumerated Classes](#)
- equals function, [Enumerated Classes](#)
- hashCode function, [Enumerated Classes](#)
- name property, [Enumerated Classes](#)
- naming conventions, [Enumerated Classes](#)
- ordinal property, [Enumerated Classes](#)
- toString function, [Enumerated Classes](#)
- valueOf function, [Enumerated Classes](#)
- values function, [Enumerated Classes](#)

equality (see [referential equality](#), [structural equality](#))

equals function

- in data classes, [equals and hashCode](#)
- in enumerated classes, [Enumerated Classes](#)
- overriding, [Operator Overloading](#), [For the More Curious: Defining Structural Comparison](#)

error function, [Preconditions](#)

error handling

(see also [exceptions](#))

in flows, [Error Handling in Flows](#)

errors

compile-time, [Compile Time vs Runtime](#)

runtime, [Compile Time vs Runtime](#)

escape character (`\`), [String Interpolation](#)

escape sequences

about, [String Interpolation](#)

ANSI, [Defining a Function That Accepts a Function](#)

exceptions

(see also [error handling](#))

about, [Exceptions](#)

`ArrayIndexOutOfBoundsException`, [Index boundaries and safe index access](#)

custom, [For the More Curious: Custom Exceptions](#)

Exception type, [Handling exceptions](#)

`IllegalArgumentException`, [Throwing an exception](#)

`IllegalStateException`, [Preconditions](#)

`NullPointerException`, [Option three: the non-null assertion operator](#)

throwing, [Throwing an exception](#)

unchecked, [For the More Curious: Checked vs Unchecked Exceptions](#)

`exitProcess` function, [Combat in NyetHack](#)

`expect` keyword, [expect and actual](#)

extensions

about, [Extensions](#)

and inheritance, [Defining an extension on a superclass](#)

bytecode representation, [Extensions, Under the Hood](#)

defining extension functions, [Defining Extension Functions](#)

extension properties, [Extension Properties](#)

for operator functions, [Operator extension functions](#)

function literals with receivers, [For the More Curious: Function Literals with Receivers](#)

generic extension functions, [Generic extension functions](#)

in the Kotlin standard library, [Extensions in the Kotlin Standard Library](#)

on nullable types, [Extensions on Nullable Types](#)

on superclasses, [Defining an extension on a superclass](#)

visibility, [Extension Visibility](#)

F

field identifier, [Property getters and setters](#)

`@file:JvmName` annotation, [Beyond Classes](#)

files

creating, [Creating your first Kotlin file](#)

running, [Running your Kotlin file](#)

filter function, [filter](#), [Flow Transformations](#)

final keyword, [Creating a Subclass](#)

first function, [Index boundaries and safe index access](#)

`flatMap` function, [Flow Transformations](#)

Float type

about, [Numeric Types](#)

`.NaN`, [Floating Point Numbers](#)

`.NEGATIVE_INFINITY`, [Floating Point Numbers](#)

`.POSITIVE_INFINITY`, [Floating Point Numbers](#)

precision, [Floating Point Numbers](#)

rounding, [Floating Point Numbers](#)

flow function, [Setting Up a Flow](#), [MutableStateFlow](#)

flows

about, [Flows](#)

cold and hot, [Setting Up a Flow](#)

collecting and emitting, [Setting Up a Flow](#)

collecting in parallel, [MutableStateFlow](#)

creating, [Setting Up a Flow](#)
error handling, [Error Handling in Flows](#)
MutableSharedFlow, [For the More Curious: SharedFlow](#)
MutableStateFlow, [MutableStateFlow](#)
operators, [Flow Termination](#)
shared, [For the More Curious: SharedFlow](#)
SharedFlow, [For the More Curious: SharedFlow](#)
StateFlow, [MutableStateFlow](#)
terminating, [Flow Termination](#)
transforming, [Flow Transformations](#)
vs channels, [Channels](#), [Joining Jobs](#)

fold function, [fold](#)

for loops, [Iteration](#)

forEach function, [Iteration](#)

forEachIndexed function, [Iteration](#)

format function, [Formatting a Double](#)

format strings, [Formatting a Double](#)

fun keyword, [Function name declaration](#)

function invocation operator (()), [Operator Overloading](#)

function types

about, [The function type](#)

type inference with, [Type Inference Support](#)

functional programming

about, [Functional Programming Basics](#)

Arrow.kt library, [For the More Curious: Arrow.kt](#)

categories of functions, [Functional Programming Basics](#)

combines, [Combining Data](#)

filters, [Filtering Data](#)

transformations, [Transforming Data](#)

functions

(see also [function types](#))

about, [Creating your first Kotlin file](#), [Functions](#)

add, [Changing a list's contents](#), [Adding elements to a set](#)

addAll, [Changing a list's contents](#), [Adding elements to a set](#)

alert, [Interacting with the DOM](#)

all, [Lambdas and the Kotlin Standard Library](#)

also, [also](#), [Using Scope Functions](#)

alternative syntax, [Single-Expression Functions](#)

and(number), [For the More Curious: Bit Manipulation](#)

anonymous (see [lambdas](#))

append, [Interacting with the DOM](#)

apply, [apply](#), [Using Scope Functions](#), [Writing Native Code with Kotlin](#)

arguments in, [Calling a Function](#)

as arguments, [Defining a Function That Accepts a Function](#)

assert, [Preconditions](#)

async, [async and await](#), [Dividing Work with Channels](#)

await, [async and await](#), [Dividing Work with Channels](#)

backtick naming syntax, [For the More Curious: Function Names in Backticks](#)

calling, [Calling a Function](#)

catch, [Error Handling in Flows](#)

chaining calls, [Generic extension functions](#)

check, [Preconditions](#)

checkNotNull, [Preconditions](#)

class functions, [Class Functions](#)

clear, [Changing a list's contents](#), [Adding elements to a set](#)

close, [Closing a Channel](#)

collect, [Setting Up a Flow](#)

combining functions, [Combining Data](#)

compareTo, [Operator Overloading](#)

contains, [Checking the contents of a list](#), [Creating a set](#), [Operator Overloading](#)

containsAll, [Checking the contents of a list](#), [Creating a set](#)
copy, [copy](#)
count, [Anonymous Functions](#)
dec, [Operator Overloading](#)
default arguments in, [Default Arguments](#)
distinct, [Collection Conversion](#)
div, [Operator Overloading](#)
drop, [Flow Termination](#)
dropWhile, [Flow Termination](#)
elementAt, [Creating a set](#)
emit, [Setting Up a Flow](#)
equals, [equals and hashCode](#), [Enumerated Classes](#), [Operator Overloading](#), [For the More Curious: Defining Structural Comparison](#)
error, [Preconditions](#)
exitProcess, [Combat in NyetHack](#)
extension functions, [Defining Extension Functions](#)
extracting using IntelliJ IDEA, [Extracting Code to Functions](#)
file-level, [For the More Curious: File-Level Functions in Java](#)
filter, [filter](#), [Flow Transformations](#)
filter functions, [Filtering Data](#)
first, [Index boundaries and safe index access](#)
flatMap, [Flow Transformations](#)
flow, [Setting Up a Flow](#), [MutableStateFlow](#)
fold, [fold](#)
forEach, [Iteration](#)
forEachIndexed, [Iteration](#)
format, [Formatting a Double](#)
from Kotlin standard library, [Scope Functions](#), [Using Scope Functions](#)
function bodies, [Anatomy of a Function](#), [Function body](#)
function headers, [Anatomy of a Function](#)
function references, [For the More Curious: Function References](#)
generateSequence, [Sequences](#)
generic extension functions, [Generic extension functions](#)
get, [Accessing a list's elements](#), [Operator Overloading](#)
getOrDefault, [Accessing Map Values](#)
getOrElse, [Index boundaries and safe index access](#), [Accessing Map Values](#)
getOrNull, [Index boundaries and safe index access](#)
getOrPut, [Adding Entries to a Map](#)
getValue, [Accessing Map Values](#)
hashCode, [equals and hashCode](#), [Enumerated Classes](#), [For the More Curious: Defining Structural Comparison](#)
higher-order, [Functional Programming Basics](#)
implicitly called, [apply](#)
in Java bytecode, [For the More Curious: File-Level Functions in Java](#)
inc, [Operator Overloading](#)
infix, [Operator extension functions](#)
inlining, [Function Inlining](#)
intArrayOf, [For the More Curious: Array Types](#)
inv(), [For the More Curious: Bit Manipulation](#)
invoke, [Operator Overloading](#)
isInitialized, [Late initialization](#)
iterator functions, [Sequences](#)
join, [Joining Jobs](#)
joinAll, [Joining Jobs](#)
js, [Executing Raw JavaScript](#)
lambdas (see [lambdas](#))
last, [Index boundaries and safe index access](#)
launch, [Coroutine Builders](#)
lazy, [Lazy initialization](#)
let, [Option two and a half: using safe calls with let](#), [let](#), [Using Scope Functions](#), [For the More Curious: Guarding Against Mutability](#)
listOf, [Lists](#)
main, [Creating your first Kotlin file](#), [Introducing NyetHack](#)

map, [Transforming Data](#), [Flow Transformations](#)

mapOf, [Creating a Map](#)

matches, [Regular Expressions](#)

measureNanoTime, [For the More Curious: Profiling](#)

measureTimeInMillis, [For the More Curious: Profiling](#)

minus, [Operator Overloading](#)

minusAssign, [Operator Overloading](#)

mutableListOf, [Changing a list's contents](#)

mutableMapOf, [Creating a Map](#)

mutableSetOf, [Adding elements to a set](#)

mutator functions, [Changing a list's contents](#), [Adding elements to a set](#)

named, [Anonymous Functions](#)

named arguments in, [Named Function Arguments](#)

naming conventions, [Function name declaration](#)

non-blocking, [Structured Concurrency](#)

none, [Lambdas and the Kotlin Standard Library](#)

onCompletion, [Flow Termination](#)

operator overloading, [Operator Overloading](#)

overloading, [For the More Curious: Function Overloading](#)

overriding, [Property getters and setters](#)

parameters, [Function parameters](#), [Calling a Function](#)

plus, [Operator Overloading](#)

plusAssign, [Operator Overloading](#)

precondition functions, [Preconditions](#)

predicate functions, [filter](#)

println, [Creating your first Kotlin file](#)

prompt, [Interacting with the DOM](#)

put, [Adding Entries to a Map](#)

putAll, [Adding Entries to a Map](#)

rangeTo, [Operator Overloading](#)

receive, [Receiving from a Channel](#), [For the More Curious: Other Channel Behaviors](#)

reduce, [reduce](#)

remove, [Changing a list's contents](#), [Adding elements to a set](#), [Adding Entries to a Map](#)

removeAll, [Changing a list's contents](#), [Adding elements to a set](#)

replace, [String Manipulation](#)

require, [Preconditions](#)

requireNotNull, [Preconditions](#)

return statements, [Function body](#), [Single-Expression Functions](#)

return types, [Function return type](#), [Single-Expression Functions](#)

roundToInt, [Converting Between Numeric Types](#)

run, [run](#), [Using Scope Functions](#)

runBlocking, [Coroutine Scopes](#)

scope, [Function scope](#)

send, [Sending to a Channel](#), [For the More Curious: Other Channel Behaviors](#)

setOf, [Creating a set](#)

shl(bitcount), [For the More Curious: Bit Manipulation](#)

shr(bitcount), [For the More Curious: Bit Manipulation](#)

single-expression, [Single-Expression Functions](#)

structure of, [Anatomy of a Function](#)

take, [Flow Termination](#)

takeIf, [takeIf](#), [Using Scope Functions](#)

takeUnless, [takeIf](#)

takeWhile, [Flow Termination](#)

test function naming, [For the More Curious: Function Names in Backticks](#)

times, [Operator Overloading](#)

to, [Creating a Map](#)

toBigDecimal, [Converting Strings to Numbers](#)

toDouble, [Converting Between Numeric Types](#), [Converting Strings to Numbers](#)

toFloat, [Converting Strings to Numbers](#)

toInt, [Converting Between Numeric Types](#), [Converting Strings to Numbers](#)

toList, [Changing a list's contents](#), [Collection Conversion](#)
toLong, [Converting Strings to Numbers](#)
toMap, [Converting Between Lists and Maps](#)
toMutableList, [Changing a list's contents](#), [Collection Conversion](#)
toMutableSet, [Collection Conversion](#)
toRegex, [Regular Expressions](#)
toSet, [Collection Conversion](#)
toString, [For the More Curious: Any](#), [toString](#), [Enumerated Classes](#)
transform, [Flow Transformations](#)
Unit functions, [Unit Functions](#)
valueOf, [Enumerated Classes](#)
values, [Enumerated Classes](#)
visibility, [Visibility modifier](#)
with, [with](#), [Using Scope Functions](#)
xor(number), [For the More Curious: Bit Manipulation](#)
zip, [zip](#), [Flow Transformations](#)

G

generateSequence function, [Sequences](#)
generic type parameters, [Defining Generic Types](#)
generics
 about, [Generics](#)
 defining a generic function, [Generic Functions](#)
 defining a generic type, [Defining Generic Types](#)
 naming conventions for type parameters, [Defining Generic Types](#)
 producers and consumers, [in and out](#)
 type constraints, [Generic Constraints](#)
 type erasure, [For the More Curious: The reified Keyword](#)
 type inference with, [Defining Generic Types](#)
get function, [Accessing a list's elements](#), [Operator Overloading](#)
get operator (see [indexed access \(get\) operator \(\[\]\)](#))
getOrDefault function, [Accessing Map Values](#)
getOrNull function, [Index boundaries and safe index access](#), [Accessing Map Values](#)
getOrPut function, [Adding Entries to a Map](#)
getValue function, [Accessing Map Values](#)
Gradle build system
 about, [Your First Kotlin Project](#)
 and project dependencies, [Enabling Coroutines](#)
greater-than operator (>), [if/else Statements](#), [Operator Overloading](#)
greater-than-or-equal-to operator (>=), [if/else Statements](#)

H

hashCode function
 in data classes, [equals and hashCode](#)
 in enumerated classes, [Enumerated Classes](#)
 overriding, [For the More Curious: Defining Structural Comparison](#)
higher-order functions, [Functional Programming Basics](#)

I

if/else statements
 comparison operators in, [if/else Statements](#)
 defining, [if/else Statements](#)
 else if branches, [Adding more conditions](#), [when Expressions](#)
 logical operators in, [Logical operators](#)
 nested, [Nested if/else statements](#)
 omitting curly braces in, [Removing braces from if/else expressions](#)

- order of conditions, [Adding more conditions](#)
- vs when expressions, [when Expressions](#)
- IllegalArgumentException, [Throwing an exception](#)
- IllegalStateException, [Preconditions](#)
- immutable, problems with terminology, [For the More Curious: Read-Only vs Immutable](#), [For the More Curious: A Closer Look at var and val Properties](#)
- imperative programming, [Why Functional Programming?](#)
- implicit returns, [Implicit returns](#)
- in keyword
 - for collections, [Operator Overloading](#)
 - for generic consumers, [in and out](#)
 - in for loops, [Iteration](#)
- in operator, for ranges, [Ranges](#)
- inc function, [Operator Overloading](#)
- increment operator (++), [Declaring a Variable](#), [Operator Overloading](#)
- indexed access (get) operator ([])
 - overloading, [Operator Overloading](#)
 - with lists, [Accessing a list's elements](#)
 - with maps, [Accessing Map Values](#)
- indices
 - forEachIndexed function, [Iteration](#)
 - in lists, [Accessing a list's elements](#)
 - out of bounds, [Index boundaries and safe index access](#)
- infix functions, [Operator extension functions](#)
- inheritance
 - about, [Inheritance](#)
 - adding functionality in subclasses, [Creating a Subclass](#)
 - and extensions, [Defining an extension on a superclass](#)
 - creating subclasses, [Creating a Subclass](#)
 - Kotlin type hierarchy, [The Kotlin Type Hierarchy](#)
 - overriding superclass functionality in subclasses, [Creating a Subclass](#)
- init keyword, [Initializer Blocks](#)
- initialization
 - about, [Initialization](#)
 - delegates, [For the More Curious: Property Delegates](#)
 - initialization order, [Initialization Order](#), [For the More Curious: Initialization Gotchas](#)
 - initializer blocks, [Initializer Blocks](#), [For the More Curious: Initialization Gotchas](#)
 - late, [Late initialization](#)
 - lazy, [Lazy initialization](#)
 - of properties, [Constructors](#)
- inline keyword, [Function Inlining](#)
- Int type
 - about, [Declaring a Variable](#), [Kotlin's Built-In Types](#), [Numeric Types](#), [Integers](#)
 - and division, [Integers](#)
 - converting from Double, [Converting Between Numeric Types](#)
 - converting to Double, [Converting Between Numeric Types](#)
- IntArray type, [For the More Curious: Array Types](#)
- intArrayOf function, [For the More Curious: Array Types](#)
- IntelliJ IDEA
 - about, [Installing IntelliJ IDEA](#), [For the More Curious: Why Use IntelliJ?](#)
 - benefits, [For the More Curious: Why Use IntelliJ?](#)
 - console, [Running your Kotlin file](#)
 - converting single-expression function syntax to block body syntax, [Throwing an exception](#)
 - creating a file, [Creating your first Kotlin file](#)
 - creating a project, [Your First Kotlin Project](#), [Your First Multiplatform Project](#)
 - displaying function return types, [Unit Functions](#)
 - editor, [Your First Kotlin Project](#)
 - error indicator, [Declaring a Variable](#)
 - extracting functions, [Extracting Code to Functions](#)
 - Gradle tool window, [Launching a Kotlin/Native application](#)
 - installing, [Installing IntelliJ IDEA](#)

Kotlin bytecode tool window, [Inspecting Kotlin Bytecode](#)

Kotlin REPL, [The Kotlin REPL](#)

launching projects through Gradle, [Launching a Kotlin/Native application](#)

opening a project, [Types](#)

overriding equals and hashCode, [For the More Curious: Defining Structural Comparison](#)

project tool window, [Your First Kotlin Project](#)

refactoring code, [Extracting Code to Functions](#)

replacing text, [expect and actual](#)

run tool window (console), [Running your Kotlin file](#)

running a project, [Running your Kotlin file](#)

Search Everywhere dialog, [Inspecting Kotlin Bytecode](#)

shortcut for adding main function, [Introducing NyetHack](#)

suspending function icon, [Structured Concurrency](#)

interface keyword, [Defining an Interface](#)

interfaces

about, [Interfaces and Abstract Classes](#)

default property getters and functions, [Default Implementations](#)

defining, [Defining an Interface](#)

implementing, [Implementing an Interface](#)

vs abstract classes, [Abstract Classes](#)

interoperating with Java

about, [Java Interoperability](#)

@file:JvmName annotation, [Beyond Classes](#)

@JvmField annotation, [Beyond Classes](#)

@JvmOverloads annotation, [Beyond Classes](#)

@JvmStatic annotation, [Beyond Classes](#)

@NotNull annotation, [Interoperability and Nullity](#)

@Nullable annotation, [Interoperability and Nullity](#)

@Throws annotation, [Exceptions and Interoperability](#)

arrays, [For the More Curious: Array Types](#)

backtick function naming syntax, [For the More Curious: Function Names in Backticks](#)

exceptions, [Exceptions and Interoperability](#)

Java classes, [Interoperating with a Java Class](#)

Java fields and properties, [Getters, Setters, and Interoperability](#)

Java method overloading, [Beyond Classes](#)

Kotlin companion objects, [Beyond Classes](#)

Kotlin default parameters, [Beyond Classes](#)

Kotlin file-level functions, [Beyond Classes](#)

Kotlin function types, [Function Types in Java](#)

Kotlin functions defined on companion objects, [Beyond Classes](#)

Kotlin lambdas, [Function Types in Java](#)

null safety, [Interoperability and Nullity](#)

platform types, [Interoperability and Nullity](#)

type mapping, [Type Mapping](#)

IntRange type, [Iteration](#)

inv() function, [For the More Curious: Bit Manipulation](#)

invoke function, [Operator Overloading](#)

iOS simulator, targeting, [Defining a Kotlin/JVM Target](#)

iOS, targeting, [Floating Point Numbers](#), [Formatting a Double](#), [Defining a Kotlin/JVM Target](#), [Kotlin/Native](#), [For the More Curious: Kotlin Multiplatform Mobile](#)
(see also [Kotlin/Native](#), [Kotlin Multiplatform](#))

is operator, [Type Checking](#)

isInitialized function, [Late initialization](#)

it identifier

with forEach, [Adding Entries to a Map](#)

with lambdas, [The it identifier](#)

with let, [let](#), [For the More Curious: Guarding Against Mutability](#)

Iterable types, [Iteration](#)

iteration

about, [Iteration](#)

break expression, [For the More Curious: The break Expression](#)

with for, [Iteration](#)
with forEach, [Iteration](#)
with forEachIndexed, [Iteration](#)
with while, [while Loops](#)

J

Java

(see also [interoperating with Java](#))
benefits of Kotlin, [Why Kotlin?](#)
checked exceptions, [For the More Curious: Checked vs Unchecked Exceptions](#)
class-level variables, [Getters, Setters, and Interoperability](#)
decompiled Kotlin bytecode, [Inspecting Kotlin Bytecode](#)
exceptions, [Exceptions and Interoperability](#)
getters and setters, [Getters, Setters, and Interoperability](#)
number formatting, [Formatting a Double](#)
package private visibility, [For the More Curious: Package Private](#)
primitive types, [For the More Curious: Java Primitive Types in Kotlin](#), [Type Mapping](#)

Java Development Kit (JDK), [Your First Kotlin Project](#)

Java Virtual Machine (JVM), targeting, [Your First Kotlin Project](#)

(see also [Kotlin/JVM](#), [Kotlin Multiplatform](#))

JavaScript, targeting, [Your First Kotlin Project](#)

(see also [Kotlin/JS](#), [Kotlin Multiplatform](#))

Job class, [Structured Concurrency](#), [Joining Jobs](#)

join function, [Joining Jobs](#)

joinAll function, [Joining Jobs](#)

js function, [Executing Raw JavaScript](#)

@JvmField annotation, [Beyond Classes](#)

@JvmOverloads annotation, [Beyond Classes](#)

@JvmStatic annotation, [Beyond Classes](#)

K

kexe (Kotlin Executable) files, [Kotlin/Native Outputs](#)

Kotlin bytecode tool window, [Inspecting Kotlin Bytecode](#)

Kotlin Executable files (kexe), [Kotlin/Native Outputs](#)

Kotlin language

documentation, [Challenge: REPL Arithmetic](#)

history, [Introducing Kotlin](#)

Kotlin Multiplatform

about, [Your First Kotlin Project](#), [What Is Kotlin Multiplatform?](#)

actual keyword, [expect and actual](#)

and application entry points, [Defining Shared Code](#)

available targets, [Defining a Kotlin/JVM Target](#)

business logic vs UI definitions, [Planning Your Multiplatform Project](#)

common code with platform-specific implementations, [expect and actual](#)

creating a project, [Your First Multiplatform Project](#)

defining a target, [Defining a Kotlin/JVM Target](#)

defining shared code, [Defining Shared Code](#)

expect keyword, [expect and actual](#)

naming conventions for files, [expect and actual](#)

shared code vs platform-specific code, [Planning Your Multiplatform Project](#)

Kotlin Multiplatform Mobile (KMM), [For the More Curious: Kotlin Multiplatform Mobile](#)

Kotlin REPL, [The Kotlin REPL](#)

Kotlin/JS

(see also [Kotlin Multiplatform](#))

about, [Your First Kotlin Project](#), [Kotlin/JS](#)

@JsName annotation, [Dynamic Types](#)

accessing HTML content from Kotlin code, [Interacting with the DOM](#)

and Kotlinx HTML library, [Interacting with the DOM](#)

calling JavaScript code from Kotlin, [The external Keyword](#)
default parameters with definedExternally, [Dynamic Types](#)
dynamic types, [Dynamic Types](#)
evaluating JavaScript with js, [Executing Raw JavaScript](#)
external keyword, [The external Keyword](#)
inspecting web page elements, [Declaring Support for Kotlin/JS](#)
number formatting, [Formatting a Double](#)
regular expressions, [Regular Expressions](#)
running projects, [Declaring Support for Kotlin/JS](#)
Safari developer tools, [Declaring Support for Kotlin/JS](#)
targeting, [Defining a Kotlin/JVM Target](#)

Kotlin/JVM

about, [Your First Kotlin Project](#)
and Java bytecode, [For the More Curious: Targeting the JVM](#), [Inspecting Kotlin Bytecode](#)
targeting, [Defining a Kotlin/JVM Target](#)

Kotlin/Native

(see also [Kotlin Multiplatform](#))
binary formats, [Declaring a macOS Target](#)
launching projects, [Launching a Kotlin/Native application](#)
numeric types, [Numeric Types](#)
regular expressions, [Regular Expressions](#)
targeting, [Defining a Kotlin/JVM Target](#), [Kotlin/Native](#)

Kotlinx libraries, [Using an HTTP Client](#)

Ktor library, [Using an HTTP Client](#)

L

lambdas

about, [Anonymous Functions](#)
and function references, [For the More Curious: Function References](#)
and Kotlin standard library functions, [Anonymous Functions](#), [Lambdas and the Kotlin Standard Library](#), [Scope Functions](#)
arguments in, [Function arguments](#)
as function arguments, [Defining a Function That Accepts a Function](#)
calling, [Lambda Expressions](#)
capturing, [For the More Curious: Capturing Lambdas](#)
defining, [Lambda Expressions](#)
implicit returns, [Implicit returns](#)
inlining, [Function Inlining](#)
it identifier, [The it identifier](#)
memory use, [Function Inlining](#)
multiple arguments in, [Accepting multiple arguments](#)
parameters, [Function arguments](#)
terminology, [Anonymous Functions](#)
trailing syntax, [Trailing lambda syntax](#)

last function, [Index boundaries and safe index access](#)

late initialization, [Late initialization](#)

lateinit keyword, [Late initialization](#)

launch function, [Coroutine Builders](#)

lazy function, [Lazy initialization](#)

lazy initialization, [Lazy initialization](#)

less-than operator (<), [if/else Statements](#)

less-than-or-equal-to operator (<=), [if/else Statements](#)

let function, [Option two and a half: using safe calls with let](#), [let](#), [Using Scope Functions](#), [For the More Curious: Guarding Against Mutability](#)

List type

(see also [lists](#))
about, [Kotlin's Built-In Types](#), [Lists](#), [Iterating Through a Map](#)
as an Iterable, [Iteration](#)
read-only vs immutable terminology, [For the More Curious: Read-Only vs Immutable](#)
vs MutableList, [Changing a list's contents](#)

listOf function, [Lists](#)

lists

(see also [List type](#), [MutableList type](#))

about, [Lists](#), [Iterating Through a Map](#)

accessing elements by index, [Accessing a list's elements](#)

checking for elements, [Checking the contents of a list](#)

converting to sequences, [Sequences](#)

converting to/from sets, [Collection Conversion](#)

creating, [Lists](#)

mutable, [Changing a list's contents](#)

read-only, [Lists](#)

var vs val, [Changing a list's contents](#)

logical operators

about, [Logical operators](#)

order of operator precedence, [Logical operators](#)

logical 'and' operator (&&), [Logical operators](#)

logical 'not' operator (!), [Logical operators](#)

logical 'or' operator (||), [Logical operators](#)

Long type, [Numeric Types](#)

M

macOS, targeting

(see also [Kotlin/Native](#), [Kotlin Multiplatform](#))

about, [Defining a Kotlin/JVM Target](#), [Kotlin/Native](#)

number formatting, [Formatting a Double](#)

numeric types, [Floating Point Numbers](#)

main function

as application entry point, [Creating your first Kotlin file](#)

IntelliJ shortcut, [Introducing NyetHack](#)

map function, [Transforming Data](#), [Flow Transformations](#)

Map type

(see also [maps](#))

about, [Kotlin's Built-In Types](#), [Maps](#)

as an Iterable, [Iteration](#)

vs MutableMap, [Adding Entries to a Map](#)

mapOf function, [Creating a Map](#)

maps

(see also [Map type](#), [MutableMap type](#))

about, [Maps](#)

accessing values by key, [Accessing Map Values](#)

adding entries, [Adding Entries to a Map](#)

creating, [Creating a Map](#)

matches function, [Regular Expressions](#)

measureNanoTime function, [For the More Curious: Profiling](#)

measureTimeInMillis function, [For the More Curious: Profiling](#)

minus assign operator (see subtraction and assignment (minus) operator (-=))

minus function, [Operator Overloading](#)

minusAssign function, [Operator Overloading](#)

mobile development, [For the More Curious: Kotlin Multiplatform Mobile](#)

modules, [Your First Kotlin Project](#), [For the More Curious: Package Private](#)

modulus/remainder operator (%), [Integers](#)

multiplatform development (see [Kotlin Multiplatform](#))

multiplication operator (*), [Operator Overloading](#)

MutableList type

(see also [lists](#))

mutator functions, [Changing a list's contents](#)

vs List, [Changing a list's contents](#)

mutableListOf function, [Changing a list's contents](#)

MutableMap type

(see also [maps](#))

- mutator functions, [Adding Entries to a Map](#)
- vs Map, [Adding Entries to a Map](#)
- mutableMapOf function, [Creating a Map](#)
- MutableSet type
 - (see also [sets](#))
 - mutator functions, [Adding elements to a set](#)
 - vs Set, [Adding elements to a set](#)
- mutableSetOf function, [Adding elements to a set](#)
- MutableSharedFlow class, [For the More Curious: SharedFlow](#)
- MutableStateFlow class, [MutableStateFlow](#)
- mutator functions
 - for lists, [Changing a list's contents](#)
 - for maps, [Adding Entries to a Map](#)
 - for sets, [Adding elements to a set](#)

N

- naming conventions
 - constants and variables, [Compile-Time Constants](#)
 - constructor parameters, [Primary constructors](#)
 - enumerated classes (enums), [Enumerated Classes](#)
 - functions, [Function name declaration](#)
 - functions, using backticks, [For the More Curious: Function Names in Backticks](#)
 - generic type parameters, [Defining Generic Types](#)
 - Kotlin Multiplatform files, [expect and actual](#)
 - Kotlin standard library files for extensions, [Extensions in the Kotlin Standard Library](#)
- native x86, targeting
 - (see also [Kotlin/Native](#))
- nested classes, [Nested Classes](#)
- non-equality operator (!=), [if/else Statements](#)
- non-null assertion (double-bang) operator (!!), [Option three: the non-null assertion operator](#)
- none function, [Lambdas and the Kotlin Standard Library](#)
- Nothing type, [For the More Curious: The Nothing Type](#)
- @NotNull annotation, [Interoperability and Nullity](#)
- null coalescing operator (?:), [The null coalescing operator](#)
- null safety
 - about, [Null Safety and Exceptions](#)
 - assert precondition function, [Preconditions](#)
 - check precondition function, [Preconditions](#)
 - checking values with != null, [Option one: checking for null values with an if statement](#)
 - checkNotNull precondition function, [Preconditions](#)
 - double-bang (non-null assertion) operator (!!), [Option three: the non-null assertion operator](#)
 - error precondition function, [Preconditions](#)
 - IllegalArgumentException, [Throwing an exception](#)
 - IllegalStateException, [Preconditions](#)
 - let function, [let](#)
 - non-null types, [Nullability](#)
 - null coalescing operator (?:), [The null coalescing operator](#)
 - nullable types, [Kotlin's Explicit Null Type](#)
 - NullPointerException, [Option three: the non-null assertion operator](#)
 - require precondition function, [Preconditions](#)
 - requireNotNull precondition function, [Preconditions](#)
 - safe call operator (?.), [Option two: the safe call operator](#)
 - try/catch statements, [Handling exceptions](#)
 - using conditionals, [Option one: checking for null values with an if statement](#)
 - when interoperating with Java, [Interoperability and Nullity](#)
- @Nullable annotation, [Interoperability and Nullity](#)
- NullPointerException, [Option three: the non-null assertion operator](#)
- numeric types
 - (see also individual types)

- about, [Numbers](#)
- bit sizes, [Numeric Types](#)
- bitwise operations, [For the More Curious: Bit Manipulation](#)
- compared, [Numeric Types](#)
- converting from strings, [Converting Strings to Numbers](#)
- for integers, [Integers](#)
- integer under/overflow, [For the More Curious: Unsigned Numbers](#)
- maximum and minimum values, [Numeric Types](#)
- signed, [Numeric Types](#)
- unsigned, [For the More Curious: Unsigned Numbers](#)

O

object declarations

- about, [Object declarations](#)
- calling, [Object declarations](#)
- initializing, [Object declarations](#)

object expressions, [Object expressions](#)

object keyword, [The object Keyword](#), [For the More Curious: Algebraic Data Types](#)

object-oriented programming

- about, [Classes](#)
- encapsulation, [Visibility and Encapsulation](#)
- inheritance, [Inheritance](#)
- polymorphism, [Creating a Subclass](#)

objects, companion, [Companion objects](#)

onCompletion function, [Flow Termination](#)

open keyword, [Creating a Subclass](#)

operator modifier, [Operator Overloading](#)

operator overloading

- about, [Operator Overloading](#)
- in extensions, [Operator extension functions](#)
- using IntelliJ IDEA generator, [For the More Curious: Defining Structural Comparison](#)

operators

- comparison, [if/else Statements](#)
- for transforming a flow, [Flow Termination](#)
- infix functions for, [Operator extension functions](#)

out keyword, [in and out](#)

override keyword

- for interface implementations, [Implementing an Interface](#)
- in subclasses, [Creating a Subclass](#)

P

packages, [Using Packages](#)

Pair type, [Creating a Map](#)

parameterized types, [Lists](#)

parameters

- (see also [functions](#))
- about, [Function parameters](#)
- in lambdas, [Function arguments](#), [Accepting multiple arguments](#)
- it identifier, [The it identifier](#)
- vs arguments, [Calling a Function](#)

platform independence, [For the More Curious: Targeting the JVM](#), [For the More Curious: Any](#)

platforms, targeting, [Challenge: REPL Arithmetic](#), [Defining a Kotlin/JVM Target](#)

- (see also individual platform names)

plus assign operator (see [addition and assignment \(plus assign\) operator \(+=\)](#))

plus function, [Operator Overloading](#)

plusAssign function, [Operator Overloading](#)

polymorphism, [Creating a Subclass](#)

precondition functions, [Preconditions](#)

primary constructors, [Primary constructors](#)
println function, [Creating your first Kotlin file](#)
private visibility, [Visibility and Encapsulation](#)
producer role, [in and out](#)
project tool window, [Your First Kotlin Project](#)
projects
 about, [Your First Kotlin Project](#)
 adding dependencies, [Enabling Coroutines](#)
 creating, [Your First Kotlin Project](#), [Your First Multiplatform Project](#)
 launching through Gradle, [Launching a Kotlin/Native application](#)
 multiplatform (see [Kotlin Multiplatform](#))
 opening, [Types](#)
 organizing with modules, [For the More Curious: Package Private](#)
 organizing with packages, [Using Packages](#)
 running, [Running your Kotlin file](#)

prompt function, [Interacting with the DOM](#)

properties
 about, [Class Properties](#), [For the More Curious: A Closer Look at var and val Properties](#)
 computed, [Computed properties](#)
 defined with val or var, [Class Properties](#), [For the More Curious: A Closer Look at var and val Properties](#)
 defining in constructors, [Defining properties in a primary constructor](#)
 dot syntax, [Class Properties](#)
 extension properties, [Extension Properties](#)
 fields, [Property getters and setters](#), [Computed properties](#)
 getters, [Property getters and setters](#)
 initializing, [Class Properties](#), [Constructors](#)
 nullable, [For the More Curious: Guarding Against Mutability](#)
 overriding getters and setters, [Property getters and setters](#)
 race conditions, [For the More Curious: Guarding Against Mutability](#)
 setters, [Property getters and setters](#)
 visibility, [Property visibility](#)
protected visibility, [Creating a Subclass](#)
public visibility, [Visibility modifier](#)
put function, [Adding Entries to a Map](#)
putAll function, [Adding Entries to a Map](#)

R

race conditions, [For the More Curious: Guarding Against Mutability](#)
range to operator (..), [Ranges](#), [Operator Overloading](#)
ranges, [Ranges](#)
rangeTo function, [Operator Overloading](#)
receive function, [Receiving from a Channel](#), [For the More Curious: Other Channel Behaviors](#)
reduce function, [reduce](#)
refactoring using IntelliJ IDEA, [Extracting Code to Functions](#)
referential equality, [String Comparison](#)
referential equality operator (==), [if/else Statements](#)
referential non-equality operator (!==), [if/else Statements](#)
reified keyword, [For the More Curious: The reified Keyword](#)
remainder/modulus operator (%), [Integers](#)
remove function, [Changing a list's contents](#), [Adding elements to a set](#), [Adding Entries to a Map](#)
removeAll function, [Changing a list's contents](#), [Adding elements to a set](#)
REPL, [The Kotlin REPL](#)
replace function, [String Manipulation](#)
require function, [Preconditions](#)
requireNotNull function, [Preconditions](#)
reserved keywords, [For the More Curious: Function Names in Backticks](#)
return keyword
 about, [Function body](#)
 absent from lambda expressions, [Implicit returns](#)

absent from Unit functions, [Unit Functions](#)
return labels, [For the More Curious: Return Labels](#)
return types, [Function return type](#), [Single-Expression Functions](#)
roundToInt function, [Converting Between Numeric Types](#)
run function, [run](#), [Using Scope Functions](#)
run tool window (console), [Running your Kotlin file](#)
runBlocking function, [Coroutine Scopes](#)
runtime errors, [Compile Time vs Runtime](#)

S

safe call operator (?.), [Option two: the safe call operator](#)
safe cast operator (as?), [For the More Curious: The Safe Cast Operator](#)
scope
 of functions, [Function scope](#)
 relative scoping, [apply](#)
scope functions, [Scope Functions](#), [Using Scope Functions](#)
sealed classes, [For the More Curious: Algebraic Data Types](#)
secondary constructors, [Secondary constructors](#)
send function, [Sending to a Channel](#), [For the More Curious: Other Channel Behaviors](#)
Sequence type
 (see also [sequences](#))
sequences
 about, [Sequences](#)
 converting from lists, [Sequences](#)
 iterator functions, [Sequences](#)
set operator ([]=), [Changing a list's contents](#)
Set type
 (see also [sets](#))
 about, [Kotlin's Built-In Types](#), [Sets](#), [Iterating Through a Map](#)
 as an Iterable, [Iteration](#)
 read-only vs immutable terminology, [For the More Curious: Read-Only vs Immutable](#)
 vs MutableSet, [Adding elements to a set](#)
setOf function, [Creating a set](#)
sets
 (see also [Set type](#), [MutableSet type](#))
 about, [Sets](#), [Iterating Through a Map](#)
 converting to/from lists, [Collection Conversion](#)
 creating, [Creating a set](#)
 index-based access, [Creating a set](#)
 mutable, [Adding elements to a set](#)
SharedFlow class, [For the More Curious: SharedFlow](#)
shl(bitcount) function, [For the More Curious: Bit Manipulation](#)
Short type, [Numeric Types](#)
shr(bitcount) function, [For the More Curious: Bit Manipulation](#)
single-expression functions, [Single-Expression Functions](#)
singletons, [The object Keyword](#)
smart casting, [Option one: checking for null values with an if statement](#), [Smart casting](#)
spread operator (*), [zip](#)
StateFlow class, [MutableStateFlow](#)
static type checking, [Declaring a Variable](#)
string concatenation, [String Interpolation](#)
string interpolation/templating
 about, [String Interpolation](#)
 interpolating an expression, [String Interpolation](#)
String type
 (see also [strings](#))
 about, [Kotlin's Built-In Types](#), [Strings](#)
strings
 about, [Strings](#)

- converting to numeric types, [Converting Strings to Numbers](#)
- immutability, [Strings are immutable](#)
- raw, [Raw Strings](#)
- structural equality, [String Comparison](#)
- structural equality operator (==), [if/else Statements](#), [Operator Overloading](#)
- structured concurrency, [Structured Concurrency](#)
- subtraction and assignment (minus assign) operator (-=)
 - about, [Declaring a Variable](#)
 - overloading, [Operator Overloading](#)
 - with lists, [Changing a list's contents](#)
 - with maps, [Adding Entries to a Map](#)
 - with set, [Adding elements to a set](#)
- subtraction operator (-), [Operator Overloading](#)
- super keyword, [Combat in NyetHack](#)
- suspend modifier, [Structured Concurrency](#)

T

- take function, [Flow Termination](#)
- takeIf function, [takeIf](#), [Using Scope Functions](#)
- takeUnless function, [takeIf](#)
- takeWhile function, [Flow Termination](#)
- this keyword
 - in class constructors, [Secondary constructors](#)
 - in extension functions, [Defining Extension Functions](#)
- throw keyword, [Throwing an exception](#), [For the More Curious: Custom Exceptions](#)
- @Throws annotation, [Exceptions and Interoperability](#)
- times function, [Operator Overloading](#)
- to function, [Creating a Map](#)
- toBigDecimal function, [Converting Strings to Numbers](#)
- toDouble function, [Converting Between Numeric Types](#), [Converting Strings to Numbers](#)
- toFloat function, [Converting Strings to Numbers](#)
- toInt function, [Converting Between Numeric Types](#), [Converting Strings to Numbers](#)
- toList function, [Changing a list's contents](#), [Collection Conversion](#)
- toLong function, [Converting Strings to Numbers](#)
- toMap function, [Converting Between Lists and Maps](#)
- toMutableList function, [Changing a list's contents](#), [Collection Conversion](#)
- toMutableSet function, [Collection Conversion](#)
- toRegex function, [Regular Expressions](#)
- toSet function, [Collection Conversion](#)
- toString function
 - about, [For the More Curious: Any](#)
 - in data classes, [toString](#)
 - in enumerated classes, [Enumerated Classes](#)
- transform function, [Flow Transformations](#)
- try/catch statements, [Handling exceptions](#)
- type casting, [Type casting](#), [For the More Curious: The Safe Cast Operator](#)
- type checking, [Declaring a Variable](#)
- type inference
 - about, [Type Inference](#)
 - with function types, [Type Inference Support](#)
 - with generics, [Defining Generic Types](#)
- types
 - (see also individual types)
 - Array, [For the More Curious: Array Types](#)
 - BigDecimal, [Floating Point Numbers](#)
 - Boolean, [Kotlin's Built-In Types](#)
 - Byte, [Numeric Types](#)
 - Char, [Kotlin's Built-In Types](#), [For the More Curious: Unicode](#)
 - collection types (see [collection types](#))

commonly used, [Kotlin's Built-In Types](#)
Double, [Kotlin's Built-In Types](#), [Numeric Types](#)
Float, [Numeric Types](#)
generic (see [generics](#))
Int, [Declaring a Variable](#), [Kotlin's Built-In Types](#), [Numeric Types](#), [Integers](#)
IntArray, [For the More Curious: Array Types](#)
IntRange, [Iteration](#)
Iterable, [Iteration](#)
Kotlin vs Java, [For the More Curious: Java Primitive Types in Kotlin](#)
List, [Kotlin's Built-In Types](#), [Lists](#), [Iterating Through a Map](#)
Long, [Numeric Types](#)
Map, [Kotlin's Built-In Types](#), [Maps](#)
MutableList, [Changing a list's contents](#)
MutableMap, [Adding Entries to a Map](#)
MutableSet, [Adding elements to a set](#)
non-null, [Nullability](#)
Nothing, [For the More Curious: The Nothing Type](#)
nullable, [Kotlin's Explicit Null Type](#)
numeric (see [numeric types](#), individual type names)
Pair, [Creating a Map](#)
platform types, [Interoperability and Nullity](#)
Sequence, [Sequences](#)
Set, [Kotlin's Built-In Types](#), [Sets](#), [Iterating Through a Map](#)
Short, [Numeric Types](#)
String, [Kotlin's Built-In Types](#), [Strings](#)
Unit, [Unit Functions](#)

U

Unicode characters, [For the More Curious: Unicode](#)
Unit type, [Unit Functions](#)

V

val keyword, [Read-Only Variables](#)
value classes, [For the More Curious: Value Classes](#)
valueOf function, [Enumerated Classes](#)
values function, [Enumerated Classes](#)
var keyword, [Declaring a Variable](#)
variables
 (see also [constants](#), [types](#))
 about, [Variables, Constants, and Types](#)
 declaring, [Declaring a Variable](#)
 file-level, [Function scope](#)
 initialization requirements, [Function scope](#)
 local, [Function scope](#)
 naming conventions, [Compile-Time Constants](#)
 read-only variables vs compile-time constants, [Compile-Time Constants](#)
 val keyword, [Read-Only Variables](#)
 var keyword, [Declaring a Variable](#)
visibility
 about, [Visibility modifier](#), [Visibility and Encapsulation](#)
 and encapsulation, [Visibility and Encapsulation](#)
 class functions, [Visibility and Encapsulation](#)
 default, [Visibility and Encapsulation](#)
 internal, [Visibility and Encapsulation](#), [For the More Curious: Package Private](#)
 modifiers, [Visibility modifier](#), [Visibility and Encapsulation](#)
 private, [Visibility and Encapsulation](#)
 properties, [Property visibility](#)
 protected, [Visibility and Encapsulation](#), [Creating a Subclass](#)

public, [Visibility and Encapsulation](#)

W

when expressions

about, [when Expressions](#)

arguments in, [when expressions without arguments](#)

scoping conditions to arguments, [when Expressions](#)

vs if/else statements, [when Expressions](#)

without arguments, [when expressions without arguments](#)

while loops, [while Loops](#)

with function, [with](#), [Using Scope Functions](#)

X

xor(number) function, [For the More Curious: Bit Manipulation](#)

Z

zip function, [zip](#), [Flow Transformations](#)