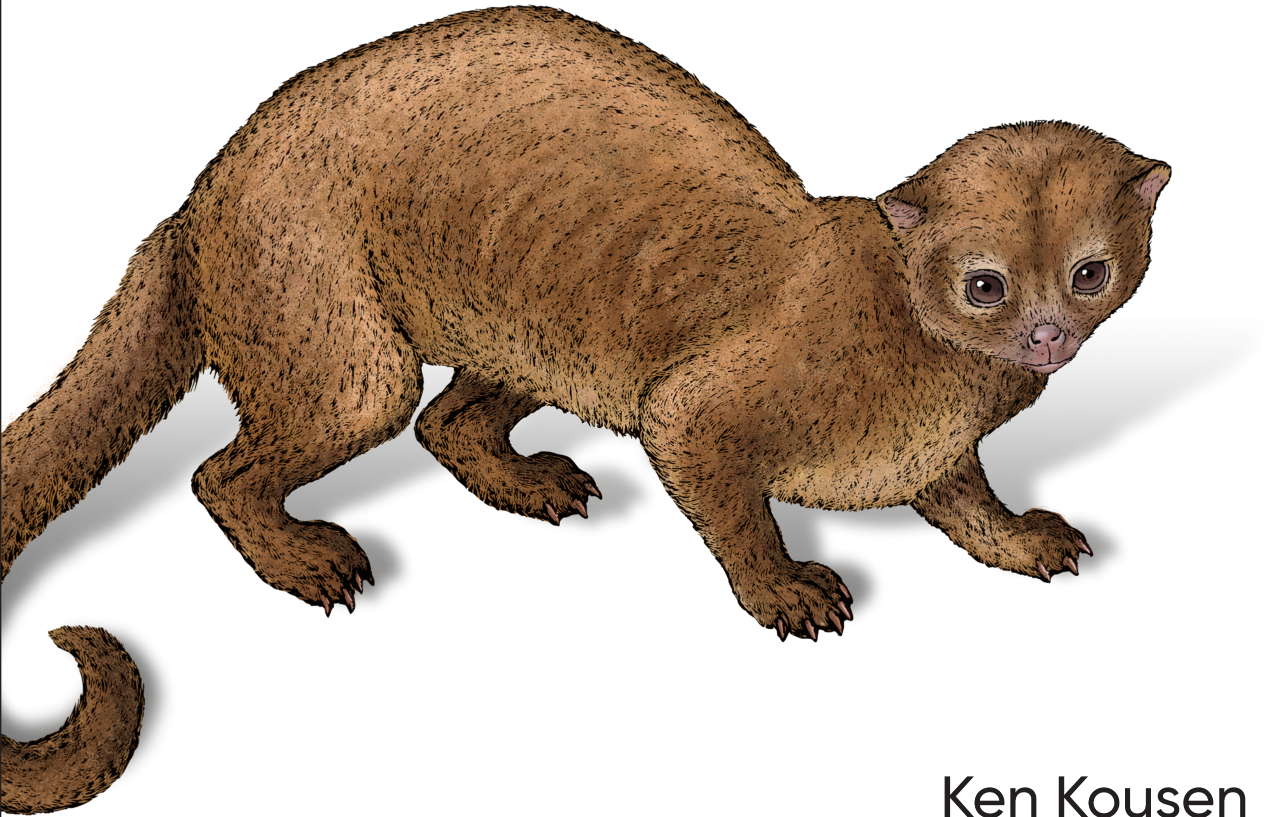
O'REILLY®

Kotlin Cookbook

A Problem-Focused Approach



Ken Kousen

O'REILLY®

Kotlin Cookbook

Use Kotlin to build Android apps, web applications, and more—while you learn the nuances of this popular language. With this unique cookbook, developers will learn how to apply this Java-based language to their own projects. Both experienced programmers and those new to Kotlin will benefit from the practical recipes in this book.

Author Ken Kousen (*Modern Java Recipes*) shows you how to solve problems with Kotlin by concentrating on your own use cases rather than on basic syntax. You provide the context and this book supplies the answers. Already big in Android development, Kotlin can be used anywhere Java is applied, as well as for iOS development, native applications, JavaScript generation, and more. Jump in and build meaningful projects with Kotlin today.

- Apply functional programming concepts, including lambdas, sequences, and concurrency
- See how to use delegates, late initialization, and scope functions
- Explore Java interoperability and access Java libraries using Kotlin
- Add your own extension functions
- Use helpful libraries such as JUnit 5
- Get practical advice for working with specific frameworks, like Android and Spring

"An excellent resource for developers looking for a quick introduction to Kotlin. The recipes provide a concise and pragmatic guide to common developer tasks while pointing out pitfalls when transitioning from Java."

—**Mark Maynard**
Senior Developer

Ken Kousen is a Java Champion, Oracle Groundbreaker Ambassador, and JavaOne Rock Star. He's the author of the books *Gradle Recipes for Android* (O'Reilly), *Making Java Groovy* (Manning), and *Modern Java Recipes* (O'Reilly) and has created O'Reilly video courses on Android, Groovy, Gradle, advanced Java, and Spring. Ken develops software and teaches software development training courses through his company, Kousen IT, Inc. He is also a JetBrains Certified Kotlin Training Partner.

PROGRAMMING LANGUAGES / JAVA

US $59.99 CAN $79.99

ISBN: 978-1-492-04667-7

5 5 9 9 9

9 781492 046677

Kotlin Cookbook

A Problem-Focused Approach

Ken Kousen

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Kotlin Cookbook
by Ken Kousen

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editors: Zan McQuade and Tyler Ortman
Development Editor: Corbin Collins
Production Editor: Christopher Faucher
Copyeditor: Sharon Wilkey
Proofreader: Charles Roumeliotis
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2019: First Edition

Revision History for the First Edition
2019-11-14: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492046677* for release details.

978-1-492-04667-7

[LSI]

For Sandra, who got me through this.

Your kindness, unflagging support, and expert skills continue to change my life.

Table of Contents

Foreword

Every few years, there is a revolutionary new language that threatens to change the way that people write software. The reality seldom lives up to the hype. Kotlin is different. Since its creation back in 2011, it has slowly, almost imperceptibly, crept its way into codebases across the world. Developers who have used Java for so long and found it lacking have been able to sneak in a little Kotlin here and there. In so doing, they have shrunk the size—and increased the power—of their code.

Having gained some fame as the preferred language for Android development, Kotlin is now at a sufficiently mature stage that a book like this is desperately needed. With a wealth of useful tips, *Kotlin Cookbook* begins at the beginning. Ken shows you how to install Kotlin and configure it for your project. He shows how to run it in a Java environment, in a browser, or as a standalone application. But the book quickly moves on, solving the kind of day-to-day programming problems faced by developers and architects everywhere.

Although there is a section set aside for Kotlin testing, you will find that the book is itself test-driven. It uses tests as practical examples of how to use the language. The tests will allow you to adapt the recipes to fit your needs more precisely.

This book brings you the kind of straightforward, practical help that will guide your progress on your Kotlin journey. It's the essential how-to Kotlin guide, and every developer should keep it on their desktop (real or virtual) to support their daily work.

Dawn and David Griffiths
Authors, *Head First Kotlin*
October 6, 2019

Preface

Welcome to *Kotlin Cookbook*! The overall focus of the book is not only to teach Kotlin syntax and semantics, but also to show you when and why a particular feature should be used. The goal isn't necessarily to cover every detail of Kotlin's syntax and libraries. In the end, however, many recipes on basic principles were added to make the book understandable even to readers with only a beginning level of Kotlin knowledge.

There is a strong movement by JetBrains to encourage the Kotlin community to embrace multiplatform, native, and JavaScript development. In the end, the decision was made *not* to include recipes involving them, since all are either in beta form or have very low adoption rates. As a result, the book concentrates exclusively on Kotlin for the JVM.

The GitHub repository for all the code can be found at *https://github.com/kousen/kotlin-cookbook*. It includes a Gradle wrapper (with the build file written in the Kotlin DSL, of course) and all the tests pass.

All of the code examples in the book have been compiled and tested with both available Long Term Support versions of Java, namely Java 8 and Java 11. Even though Java 8 is technically past its end-of-life deadline, it is still pervasive enough in the industry to ensure the code examples work with it. At the time of this writing, the current version of Kotlin is 1.3.50, with 1.3.60 on the way. All the code works with both versions, and the GitHub repository will frequently be updated to use the latest version of Kotlin.

Who Should Read This Book

This book is written for developers who already know the basics of object-oriented programming, especially in Java or another JVM-based language. While Java knowledge would be helpful, it isn't required.

A recipe book like this one is more focused on using the techniques and idioms of Kotlin than on being an exhaustive resource on the language. That has the advantage of using the full power of the language in any given recipe, but the disadvantage of spending only a limited time on the basics of those features. Each chapter includes a summary of the basic techniques, so if you are only vaguely familiar with how to create collections, work with arrays, or design classes, you should still be fine. The online reference manual provides a solid introduction to the language, and the book makes frequent reference to examples and discussions found there.

In addition, the book frequently dives into the implementations of features from the Kotlin libraries. That's to show how the developers of the language work with it in practice, as well as to discuss why things are done the way they are. No prior knowledge of the implementation is expected, however, and you are free to skip those details if you are in a hurry.

How This Book Is Organized

This book is organized into recipes, and while each is self-contained, many reference others in the book. The hope is that you can read them in any particular order. That said, there is a loose ordering to the chapters, as follows:

- Chapter 1 covers the basic process of installing and running Kotlin, including using the REPL, working with build tools like Maven and Gradle, and employing the native image generator in Graal.
- Chapter 2 covers some fundamental features of Kotlin—such as nullable types, overloading operators, and converting between types—before examining some more esoteric issues including working with bitwise shift operators or the `to` extension function on the `Pair` class.
- Chapter 3 focuses on object-oriented features of the language that developers from other languages might find surprising or unusual. It includes how to use the `const` keyword, how Kotlin handles backing properties, delayed initialization, and the dreaded `Nothing` class, which is guaranteed to confuse existing Java developers.
- Chapter 4 has only a few recipes, which involve functional features that need their own explanations. Functional programming concepts are covered throughout the book, especially when talking about collections, sequences, and coroutines, but there are a handful of techniques included in this chapter that you may find unusual or particularly interesting.
- Chapter 5 covers arrays and collections, dealing mostly with nonobvious methods like destructing collections, sorting by multiple properties, building a window on a collection, and creating progressions.

- Chapter 6 shows how Kotlin handles sequences of items lazily, similar to the way Java uses streams. Recipes cover generating sequences, yielding from them, and working with infinite sequences.
- Chapter 7 covers another topic unique to Kotlin: functions that execute a block of code in the context of an object. Functions like `let`, `apply`, and `also` are quite useful in Kotlin, and this chapter illustrates why and how to use them.
- Chapter 8 discusses a convenient feature of Kotlin: how it implements delegation. Delegation lets you employ composition rather than inheritance, and Kotlin includes several delegates in the standard library, like `lazy`, `observable`, and `vetoable`.
- Chapter 9 covers the important topic of testing, with a particular focus on JUnit 5. In its current version, JUnit is designed to work well with Kotlin, and that includes both its regular usage and employing it in Spring Framework applications. This chapter discusses several approaches that make writing and executing tests easier.
- Chapter 10 includes a couple of recipes specifically for managing resources. File I/O is covered, as is the `use` function, which has broad applicability in several contexts.
- Chapter 11 covers topics that do not fit easily in any other category. Topics such as how to get the current Kotlin version, how to force the `when` statement to be exhaustive even when it doesn't return a value, and how to use the `replace` function with regular expressions are covered. In addition, the `TODO` function and the `Random` class are discussed, as well as how to integrate with Java exception handling.
- Chapter 12 involves the Spring Framework along with Spring Boot, which is very friendly to Kotlin. A few recipes are included to show how to use Kotlin classes as managed beans, how to implement JPA persistence, and how to inject dependencies when needed.
- Chapter 13 covers the subject of coroutines, one of the most popular features of Kotlin and the basis of concurrent and parallel programming in the language. Recipes cover the basics, like builders and dispatchers, along with how to cancel and debug coroutines, and how to run them on your own custom Java thread pool.

The chapters, and indeed the recipes themselves, do not have to be read in any particular order. They do complement each other, and each recipe ends with references to others, but you can start reading anywhere. The chapter groupings are provided as a way to put similar recipes together, but it is expected that you will jump from one to another to solve whatever problem you may have at the moment.

Special note for Android developers: Kotlin is now the preferred language for Android development, but it is a much broader, general-purpose programming language. You can use it anywhere you would use Java, and more. This book does not have a dedicated section just for Android. Instead, Android uses of Kotlin are discussed throughout. A few specific Android-related recipes, like coroutine cancellation, take advantage of the fact that Android libraries make extensive use of Kotlin, but in general the features of the language covered in this book can be used anywhere. It is hoped that by covering the language in a more general way, Android developers will find techniques useful to them in any coding project.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/kousen/kotlin-cookbook*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not generally require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Kotlin Cookbook* by Ken Kousen (O'Reilly). Copyright 2020 Ken Kousen, 978-1-492-04667-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information: *https://oreil.ly/kotlin-cookbook*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

At the Google I/O conference in 2017, the company announced that Kotlin would be a supported language for Android development. Later that year, Gradle, Inc.—the company behind the Gradle build tool—announced that it would support a Gradle domain-specific language (DSL) for builds. Both of those developments convinced me to dig into the language, and I've been happy to have done so.

Over the past few years, I've been giving regular presentations and workshops on Kotlin. While the basics of the language are easy to learn and apply, I've been impressed with its depth and how aware it is of the way modern development ideas are implemented in other languages, like Groovy or Scala. Kotlin is a synthesis of many of the best programming ideas throughout the industry, and I've learned a lot by doing the deep dive necessary to write a book like this.

As part of my learning process, I've benefitted from working with many active Kotlin developers, including Dawn and Dave Griffiths, whose books *Head First Android Development* and *Head First Kotlin* are outstanding; they even agreed to write the foreword for this book. Hadi Harriri, a developer advocate at JetBrains, gives presentations on Kotlin on a regular basis. Those talks always inspire me to spend time on the language, and he was kind enough to be a technical reviewer for this book. I'm very grateful to them.

Bill Fly also provided a technical review. I've interacted with him on the O'Reilly Learning Platform more times than I can count, and he always provides interesting insights (and hard questions). My good friend Jim Harmon helped me get up to speed on Android many years ago, and has always been willing to answer my questions and talk about how Kotlin is used in practice. Mark Maynard is an active developer in industry who helped me understand how Kotlin worked with the Spring Framework, and I'm grateful for that. Finally, the inimitable Venkat Subramaniam was kind enough to take time from his busy schedule writing his own Kotlin book (entitled *Programming Kotlin*: it's as good as the rest of his books) to help me with mine. I'm happy to know all my tech reviewers and am humbled by the amount of time and effort they spent improving the book you see now.

I need to acknowledge many of my fellow speakers on the NFJS tour, including Nate Schutta, Michael Carducci, Matt Stine, Brian Sletten, Mark Richards, Pratik Patel, Neal Ford, Craig Walls, Raju Gandhi, Jonathan Johnson, and Dan "the Man" Hinojosa for their constant doses of perspective and encouragement. I'm sure I've left out someone on the tour, and, if so, I assure you it was deliberate.

Okay, maybe not. Writing books and teaching training classes (my actual day job) are solitary pursuits. It's great having a community of friends and colleagues that I can rely on for perspective, advice, and various forms of entertainment.

Many people at O'Reilly Media were involved in the creation of this book. Rather than call them out individually, I specifically want to mention Zan McQuade, who was frequently placed in awkward positions by my irregular schedule and my general contrary nature. Thank you for your patience, understanding, and hard work to bring the book to completion.

Finally, I need to express all my love to my wife, Ginger, and my son, Xander. Without the support and kindness of my family, I would not be the person I am today, a fact that grows more obvious to me with each passing year. I can never express what you both mean to me.

CHAPTER 1

Installing and Running Kotlin

The recipes in this chapter help you get up and running with the Kotlin compiler, both from the command line and using an integrated development environment (IDE).

1.1 Running Kotlin Without a Local Compiler

Problem

You want to try out Kotlin without a local installation, or run it on a machine that does not support it (for example, a Chromebook).

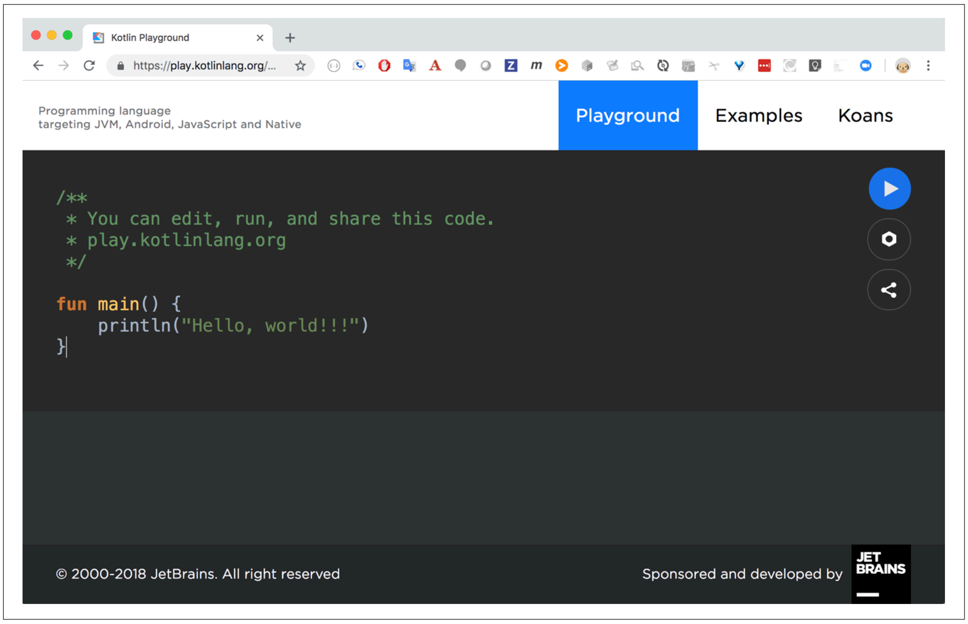
Solution

Use the Kotlin Playground, an online sandbox for exploring Kotlin.

Discussion

The *Kotlin Playground* provides an easy way to experiment with Kotlin, explore features you haven't used, or simply run Kotlin on systems that don't have an installed compiler. It gives you access to the latest version of the compiler, along with a web-based editor that allows you to add code without installing Kotlin locally.

Figure 1-1 is a snapshot of the browser page.


Kotlin Playground
https://play.kotlinlang.org/...
Programming language
targeting JVM, Android, JavaScript and Native
Playground
Examples
Koans

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */

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

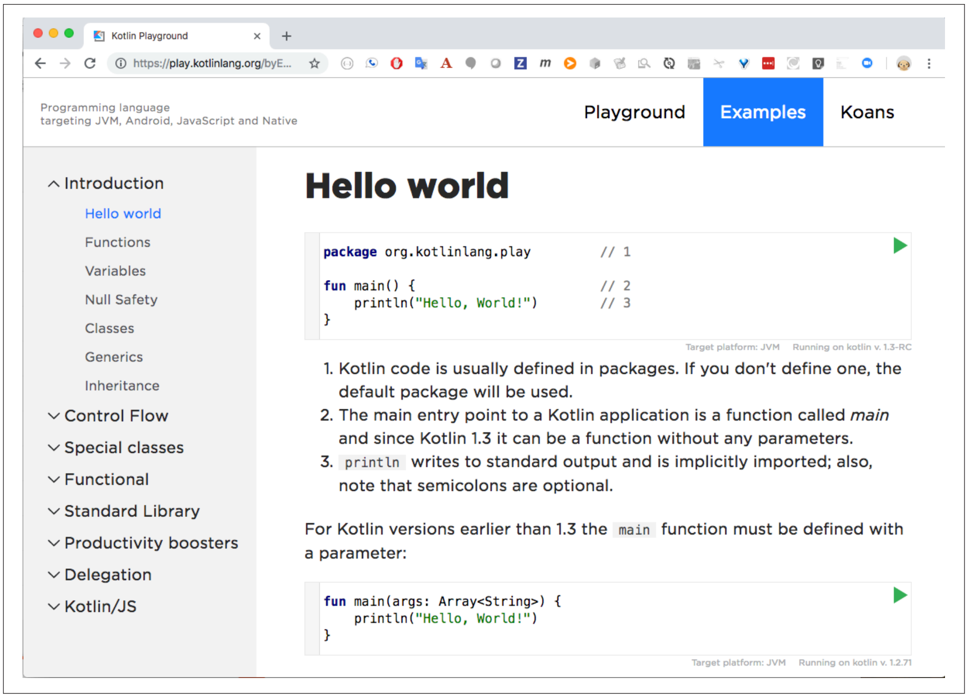
© 2000-2018 JetBrains. All right reserved
Sponsored and developed by
JET BRAINS

Figure 1-1. The Kotlin Playground home page

Just type in your own code and click the Play button to execute it. The Settings button (the gear icon) allows you to change Kotlin versions, decide which platform to run on (JVM, JS, Canvas, or JUnit), or add program arguments.

> As of Kotlin 1.3, the Kotlin function `main` can be defined without parameters.

The Examples section contains an extensive set of sample programs, organized by topic, that can be executed using an embedded block in a browser. Figure 1-2 shows the “Hello world” program page.



Hello world

```
package org.kotlinlang.play         // 1

fun main() {                        // 2
    println("Hello, World!")        // 3
}
```

1. Kotlin code is usually defined in packages. If you don't define one, the default package will be used.
2. The main entry point to a Kotlin application is a function called *main* and since Kotlin 1.3 it can be a function without any parameters.
3. `println` writes to standard output and is implicitly imported; also, note that semicolons are optional.

For Kotlin versions earlier than 1.3 the `main` function must be defined with a parameter:

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

Figure 1-2. Examples in the Kotlin Playground

The playground also has a dedicated section for *Kotlin Koans*, which are a series of exercises to help you become more familiar with the language. While these are useful online, if you use IntelliJ IDEA or Android Studio, the Koans can be added using the EduTools plug-in.

1.2 Installing Kotlin Locally

Problem

You want to execute Kotlin from a command prompt on your local machine.

Solution

Perform a manual install from GitHub or use one of the available package managers for your operating system.

Discussion

The page at *http://kotlinlang.org/docs/tutorials/command-line.html* discusses the options for installing a command-line compiler. One option is to download a ZIP file containing an installer for your operating system. This page contains a link to the GitHub repository for Kotlin current releases. ZIP files are available for Linux, macOS, Windows, and the source distribution. Simply unzip the distribution and add its *bin* subdirectory to your path.

A manual install certainly works, but some developers prefer to use package managers. A *package manager* automates the installation process, and some of them allow you to maintain multiple versions of a particular compiler.

SDKMAN!, Scoop, and other package managers

One of the most popular installation programs is SDKMAN!. Originally designed for Unix-based shells, there are plans to make it available for other platforms as well.

Installing Kotlin with SDKMAN! begins with a `curl` install:

```
> curl -s https://get.sdkman.io | bash
```

Then, once it's installed, you can use the `sdk` command to install any one of a variety of products, including Kotlin:

```
> sdk install kotlin
```

By default, the latest version will be installed in the *~/.sdkman/candidates/kotlin* directory, along with a link called `current` that points to the selected version.

You can find out what versions are available by using the `list` command:

```
> sdk list kotlin
```

The `install` command by default selects the latest version, but the `use` command will let you select any version, offering to install it if necessary:

```
> sdk use kotlin 1.3.50
```

That will install version 1.3.50 of Kotlin, if necessary, and use it in the current shell.

IntelliJ IDEA or Android Studio can use the downloaded versions, or they can maintain their own versions separately.

Other package managers that support Kotlin include Homebrew, MacPorts, and Snapcraft.

On Windows, you can use Scoop. Scoop does for Windows what the other package managers do for non-Windows systems. Scoop requires PowerShell 5 or later and .NET Framework 4.5 or later. Simple installation instructions are found on the Scoop website.

Once Scoop is installed, the *main* bucket allows you to install the current version of Kotlin:

```
> scoop install kotlin
```

This will install the scripts *kotlin.bat*, *kotlinc.bat*, *kotlin-js.bat*, and *kotlin-jvm.bat* and add them all to your path.

That is sufficient, but if you want to try it, there is an experimental installer called `kotlin-native`, which installs a native Windows compiler as well. This installs an LLVM backend for the Kotlin compiler, a runtime implementation, and a native code generation facility by using the LLVM toolchain.

Regardless of how you install Kotlin, you can verify that it works and is in your path by using the simple command `kotlin -version`. A typical response to that command is shown here:

```
> kotlin -version
Kotlin version 1.3.50-release-112 (JRE 13+33)
```

See Also

Recipe 1.3 discusses how to use Kotlin from the command line after it is installed.

1.3 Compiling and Running Kotlin from the Command Line

Problem

You want to compile and execute Kotlin from the command line.

Solution

Use the `kotlinc-jvm` and `kotlin` commands, similar to Java.

Discussion

The Kotlin SDK for the JVM includes the Kotlin compiler command, `kotlinc-jvm`, and the Kotlin execution command, `kotlin`. They are used just like `javac` and `java` for Java files.

The Kotlin installation includes a script called *kotlinc-js* for compiling to JavaScript. This book assumes you are planning to use the JVM version. The basic script *kotlinc* is an alias for *kotlinc-jvm*.

For example, consider a trivial "Hello, Kotlin!" program, stored in a file called *hello.kt*, with the code shown in Example 1-1.

Example 1-1. hello.kt

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

The command `kotlinc` compiles this file, and the command `kotlin` is used to execute the resulting class file, as in Example 1-2.

Example 1-2. Compiling and executing a regular Kotlin file

```
> kotlinc-jvm hello.kt                ❶

> ls
hello.kt HelloKt.class

> kotlin HelloKt                  ❷
Hello, Kotlin!
```

❶ Compiles the source

❷ Executes the resulting class file

The compiler produces the *HelloKt.class* file, which contains bytecodes that can be executed on the Java Virtual Machine. Kotlin does not generate Java source code—it's not a transpiler. It generates bytecodes that can be interpreted by the JVM.

The compiled class takes the name of the file, capitalizes the first letter, and appends *Kt* on the end. This can be controlled with annotations.

If you wish to produce a self-contained JAR file that can be executed by the Java command, add the `-include-runtime` argument. That allows you to produce an executable JAR that can be run from the `java` command, as in Example 1-3.

Example 1-3. Including the Kotlin runtime

```
> kotlinc-jvm hello.kt -include-runtime -d hello.jar
```

The resulting output file is called *hello.jar*, which can be executed using the `java` command:

```
> java -jar hello.jar
Hello, Kotlin!
```

Leaving out the `-include-runtime` flag would produce a JAR file that needs the Kotlin runtime on the classpath in order to execute.

> The `kotlinc` command without any arguments starts the interactive Kotlin REPL, which is discussed in Example 1-4.

See Also

Example 1-4 shows how to use the Kotlin read-eval-print loop (REPL) for interactive coding. Recipe 1.5 discusses executing Kotlin scripts from the command line.

1.4 Using the Kotlin REPL

Problem

You want to run Kotlin in an interactive shell.

Solution

Use the Kotlin REPL by typing `kotlinc` by itself at the command line.

Discussion

Kotlin includes an interactive compiler session manager, known as a REPL (read-eval-print loop) that is triggered by the `kotlinc` command with no arguments. Once inside the REPL, you can evaluate arbitrary Kotlin commands and see the results immediately.

> The Kotlin REPL is also available inside Android Studio and IntelliJ IDEA as the Kotlin REPL entry under the Tools → Kotlin menu.

After running the `kotlinc` command, you will receive an interactive prompt. An example session is shown in Example 1-4.

Example 1-4. Using the Kotlin REPL

```
▶ kotlinc
Welcome to Kotlin version 1.3.50 (JRE 11.0.4+11)
Type :help for help, :quit for quit
>>> println("Hello, World!")
Hello, World!
>>> var name = "Dolly"
>>> println("Hello, $name!")
Hello, Dolly!

>>> :help
Available commands:
:help                   show this help
:quit                   exit the interpreter
:dump bytecode          dump classes to terminal
:load <file>            load script from specified file

>>> :quit
```

The REPL is a quick and easy way to evaluate Kotlin expressions without starting up a full IDE. Use it if you don't want to create an entire project or other IDE-based collection of files, or if you want to do a quick demo in order to help another developer, or if you don't have your preferred IDE available.

1.5 Executing a Kotlin Script

Problem

You want to write and execute a script written in Kotlin.

Solution

Enter your code in a file ending in *.kts*. Then use the `kotlinc` command with the `-script` option to execute it.

Discussion

The `kotlinc` command supports several command-line options, one of which allows Kotlin to be used as a scripting language. A *script* is defined as a Kotlin source file with the file extension *.kts* that includes executable code.

As a simple example, the file *southpole.kts* in Example 1-5 shows the current time at the South Pole and prints whether it is currently on daylight saving time. The script uses the *java.time* package added to Java in version 8.

Example 1-5. southpole.kts

```
import java.time.*

val instant = Instant.now()
val southPole = instant.atZone(ZoneId.of("Antarctica/South_Pole"))
val dst = southPole.zone.rules.isDaylightSavings(instant)
println("It is ${southPole.toLocalTime()} (UTC${southPole.offset})
        at the South Pole")
println("The South Pole ${if (dst) "is" else "is not"} on Daylight Savings Time")
```

Execute this file with the `kotlinc` command using the `-script` option:

```
> kotlinc -script southpole.kts
It is 10:42:56.056729 (UTC+13:00) at the South Pole
The South Pole is on Daylight Savings Time
```

Scripts contain the code that would normally appear inside the standard `main` method in a class. Kotlin can be used as a scripting language on the JVM.

1.6 Building a Standalone Application Using GraalVM

Problem

You want to generate an application that can be run from the command line without any additional dependencies.

Solution

Use the GraalVM compiler and `native-image` tool.

Discussion

GraalVM is a high-performance virtual machine that provides a cross-language runtime for running applications written in a variety of languages. You can write in a JVM-based language like Java or Kotlin, and integrate with JavaScript, Ruby, Python, R, and more.

One nice feature of GraalVM is that you can use it to create a native executable from your code. This recipe shows a simple example of how to use GraalVM's `native-image` tool to create a native binary from Kotlin source code.

You can install GraalVM from the downloads page. For the current recipe, the free Community Edition was installed using the SDKMAN! tool:

```
> sdk install java 19.2.0.1-grl
> java -version
openjdk version "1.8.0_222"
OpenJDK Runtime Environment (build 1.8.0_222-20190711112007.graal.jdk8u-src...
OpenJDK 64-Bit GraalVM CE 19.2.0.1 (build 25.222-b08-jvmci-19.2-b02, mixed mode)

> gu install native-image
// installs native image component
```

Consider the Kotlin version of “Hello, World!” from Figure 1-1, reproduced here:

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

As stated in Recipe 1.3, you can just compile this script by using `kotlinc-jvm`, which generates *HelloKt.class*, and then run with `kotlin`:

```
> kotlinc-jvm hello.kt  // generates HelloKt.class
> kotlin HelloKt
Hello, World!
```

To generate a native image instead, first compile the script with the `-include-runtime` option. That generates a file called *hello.jar*:

```
> kotlinc-jvm hello.kt -include-runtime -d hello.jar
```

The GraalVM system includes the `native-image` tool, which you can use to generate the native executable, as in Example 1-6.

Example 1-6. Building a native executable using GraalVM

```
> native-image -jar hello.jar
```

From the docs: “For compilation, *native-image* depends on the local toolchain, so please make sure *glibc-devel*, *zlib-devel* (header files for the C library and zlib) and *gcc* are available on your system.”

The output will resemble the following:

```
> native-image -jar hello.jar
Build on Server(pid: 61247, port: 49590)*
[hello:61247]    classlist:   1,497.63 ms
[hello:61247]        (cap):   2,225.47 ms
[hello:61247]        setup:   3,451.98 ms
[hello:61247]   (typeflow):   2,163.16 ms
[hello:61247]    (objects):   1,793.53 ms
[hello:61247]   (features):     215.90 ms
[hello:61247]     analysis:   4,247.68 ms
[hello:61247]     (clinit):     107.96 ms
[hello:61247]     universe:     399.58 ms
[hello:61247]      (parse):     329.84 ms
[hello:61247]     (inline):     753.12 ms
[hello:61247]    (compile):   3,426.14 ms
```

```
    [hello:61247]      compile:   4,807.54 ms
    [hello:61247]        image:     306.96 ms
    [hello:61247]        write:     180.22 ms
    [hello:61247]      [total]:  15,246.88 ms
```

The result will be a file called *hello* that you can invoke at the command line. On a Mac or other Unix-based system, you'll see the following:

```
    > ./hello
    Hello, World!
```

There are now three ways to run the original script:

- Compile with `kotlinc-jvm` and then execute with `kotlin`.
- Compile including the runtime and then execute the resulting JAR with `java`.
- Compile with `kotlinc`, create a native image with GraalVM, and then execute from the command line.

The sizes of the resulting files are quite different. The compiled bytecode file *HelloKt.class* is only about 700 bytes. The *hello.jar* file with its included runtime is about 1.2 MB. The native image is still larger, at about 2.1 MB. The speed differences are dramatic however, even on a tiny script like this. Example 1-7 shows a simple comparison.

Example 1-7. Timing the hello script

```
> time kotlin HelloKt
Hello, World!
kotlin HelloKt  0.13s user 0.05s system 112% cpu 0.157 total

~/Documents/kotlin
> time java -jar hello.jar
Hello, World!
java -jar hello.jar  0.08s user 0.02s system 99% cpu 0.106 total

~/Documents/kotlin
> time ./hello
Hello, World!
./hello  0.00s user 0.00s system 59% cpu 0.008 total
```

The relative values are telling. While the JAR file is somewhat quicker than running `kotlin` directly, the native image is literally an order of magnitude faster. In this example, it takes only about 8 milliseconds to run.

If you are a Gradle user, you can use a GraalVM plug-in called `gradle-graal`. It adds a `native-image` task (among others) to your build. See the home page for the plug-in for details.

1.7 Adding the Kotlin Plug-in for Gradle (Groovy Syntax)

Problem

You want to use add the Kotlin plug-in to a Gradle build by using the Groovy domain-specific language (DSL) syntax.

Solution

Add the Kotlin dependency and plug-in by using the Groovy DSL tags in a build file.

Discussion

This recipe uses the Groovy DSL for Gradle. The next recipe shows how to use the Kotlin DSL for Gradle instead.

The Gradle build tool supports compiling Kotlin on the JVM by using a plug-in supplied by JetBrains. The `kotlin-gradle-plugin, adding to Gradle build script` has been registered at the Gradle plug-ins repository, and can be added to a Gradle build script as in Example 1-8. The code shown is added to a file called *build.gradle* in the project root.

Example 1-8. Adding the Kotlin plug-in by using the `plugins` block (Groovy DSL)

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.3.50"
}
```

The `version` value represents both the plug-in version and the Kotlin version. Gradle still supports the older syntax for adding plug-ins, as shown in Example 1-9.

Example 1-9. Older syntax for the Kotlin plug-in (Groovy DSL)

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:1.3.50'
    }
}

plugins {
    id "org.jetbrains.kotlin.jvm" version "1.3.50"
}
```

Both of these snippets use the Groovy DSL syntax for Gradle build files, which supports both single- and double-quoted strings. As with Kotlin, Groovy uses double-quoted strings when doing interpolation, but since none is required here, single-quoted strings also work.

The `plugins` block does not require you to state where the plug-in is found, as in the `repositories` block in the latter example. This is true for any Gradle plug-in registered at the Gradle plug-ins repository. Using the `plugins` block also automatically "applies" the plug-in, so no `apply` statement is required either.

The *settings.gradle* file is recommended but not required. It is processed during the initialization phase of Gradle processing, which occurs when Gradle determines which project build files need to be analyzed. In a multiproject build, the settings file shows which subdirectories of the root are themselves Gradle projects as well. Gradle can share settings and dependencies among subprojects, can make one subproject depend on another, or can even process subproject builds in parallel. For details, see the multiproject build sections of the Gradle User Manual.

Kotlin sources can be mixed with Java sources in the same folder, or you can create separate *src/main/java* and *src/main/kotlin* folders for them individually.

Android projects

The Kotlin plug-in for Android is handled slightly differently. Android projects are multiproject builds in Gradle, meaning they normally have two *build.gradle* files: one in the root directory, and one in a subdirectory called *app* by default. Example 1-10 shows a typical top-level *build.gradle* file containing only the Kotlin plug-in information.

Example 1-10. Using Kotlin in Android projects (Groovy DSL)

```
buildscript {
    ext.kotlin_version = '1.3.50'
    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.0'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

// ... more tasks, unrelated to the plug-in ...
```

In Gradle parlance, the plug-in is then *applied*, as in the typical *app* directory *build.gradle* file shown in Example 1-11.

Example 1-11. Applying the Kotlin plug-in

```
apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'            ❶

apply plugin: 'kotlin-android-extensions' ❷

android {
    // ... android information ...
}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version" ❸

    // ... other unrelated dependencies ...
}
```

❶ Applies the Kotlin plug-in for Android

❷ Applies the Android Kotlin extensions

❸ Standard library dependency, can use JDK 8 or JDK 7

The Kotlin plug-in for Android is declared in the `buildscript` section and then applied in this file. The plug-in knows how to compile Kotlin code inside your Android application. The downloaded plug-in also includes the Android extensions, which makes it easy to access Android widgets by their ID values.

The Kotlin plug-in can generate bytecodes for either JDK 7 or JDK 8. Change the `jdk` value in the listed dependency to select whichever you prefer.

At the time of this writing, there is no option to select the Kotlin DSL when creating Android projects. You can create your own build files that use the Kotlin DSL, but that is unusual. The Kotlin DSL will be available in version 4.0 of Android Studio, which will also include full support for KTS files and Kotlin live templates.

See Also

The same process using the Kotlin DSL is shown in Recipe 1.8, other than for the Android section.

1.8 Adding the Kotlin Plug-in for Gradle (Kotlin Syntax)

Problem

You want to add the Kotlin plug-in to a Gradle build, using the Kotlin DSL.

Solution

Add the Kotlin dependency and plug-in, using the Kotlin DSL tags in a build file.

Discussion

This recipe uses the Kotlin DSL for Gradle. The previous recipe shows how to use the Groovy DSL for Gradle instead.

Gradle (version 5.0 and above) includes the new Kotlin DSL for configuring the build file. It also makes available `kotlin-gradle-plugin`, registered at the Gradle plug-in repository, which can be added to a Gradle build script shown in Example 1-12. Alternatively, you can use the older `buildscript` syntax in Example 1-13. The code shown is added to a file called *build.gradle.kts* in the project root.

Example 1-12. Adding the Kotlin plug-in by using the `plugins` block (Kotlin DSL)

```
plugins {
    kotlin("jvm") version "1.3.50"
}
```

Example 1-13. Older syntax for the Kotlin plug-in (Kotlin DSL)

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath(kotlin("gradle-plugin", version = "1.3.50"))
    }
}
```

```
plugins {
    kotlin("jvm")
}
```

The `plugins` block does not require you to state where the plug-in is found, as in the `repositories` block in the latter example. This is true for any Gradle plug-in registered at the Gradle plug-ins tepository. Using the `plugins` block also automatically "applies" the plug-in, so no `apply` statement is required either.

The default build filenames for the Kotlin DSL in Gradle are *settings.gradle.kts* and *build.gradle.kts*.

As you can see, the biggest differences from the Groovy DSL syntax are as follows:

- Double quotes are required on any strings.
- The parentheses are required in the Kotlin DSL.
- Kotlin uses an equals sign (`=`) to assign values, rather than a colon (`:`).

The *settings.gradle.kts* file is recommended but not required. It is processed during the initialization phase of Gradle processing, which occurs when Gradle determines which project build files need to be analyzed. In a multiproject build, the settings file shows which subdirectories of the root are themselves Gradle projects as well. Gradle can share settings and dependencies among subprojects, can make one subproject depend on another, or can even process subproject builds in parallel. For details, see the multiproject build sections of the Gradle User Manual.

Kotlin and Java source code can be mixed together in *src/main/java* or *src/main/kotlin*, or you can add your own source files by using the `sourceSets` property of Gradle. See the Gradle documentation for details.

See Also

The same process using the Groovy DSL in Gradle is shown in Recipe 1.7. Additional details for Android projects can be found in that recipe as well, as the Kotlin DSL is not currently an option when generating Android projects.

1.9 Using Gradle to Build Kotlin Projects

Problem

You want to build a project that contains Kotlin by using Gradle.

Solution

In addition to the Kotlin plug-in for Gradle, add the Kotlin JDK dependency at compile time.

Discussion

The examples in Recipes 1.7 and 1.8 showed how to add the Kotlin plug-in for Gradle. This recipe adds features to the build file to process any Kotlin code in your project.

To compile the Kotlin code in Gradle, you need to add an entry to the `dependencies` block in your Gradle build file, as shown in Example 1-14.

Example 1-14. Kotlin DSL for simple Kotlin project (build.gradle.kts)

```
plugins {
    `java-library`                      ❶
    kotlin("jvm") version "1.3.50"      ❷
}

repositories {
    jcenter()
}

dependencies {
    implementation(kotlin("stdlib"))    ❸
}
```

❶ Adds tasks from the Java Library plug-in

❷ Adds the Kotlin plug-in to Gradle

❸ Adds the Kotlin standard library to the project at compile time

The `java-library` plug-in defines tasks for a basic JVM-based project, like `build`, `compileJava`, `compileTestJava`, `javadoc`, `jar`, and more.

The `plugins` section must come first, but the other top-level blocks (`repositories`, `dependencies`, etc.) can be in any order.

The `dependencies` block indicates that the Kotlin standard library is added at compile time (older versions of Gradle still use the `compile` configuration instead of `implementation`, but the effect is the same). The `repositories` block indicates that the Kotlin dependency will be downloaded from `jcenter`, which is the public Artifactory Bintray repository.

If you run the `gradle build --dry-run` task at the command line, you can see the tasks that would be executed by Gradle without actually running them. The result is as follows:

```
> gradle build -m

:compileKotlin SKIPPED
:compileJava SKIPPED
:processResources SKIPPED
:classes SKIPPED
:inspectClassesForKotlinIC SKIPPED
:jar SKIPPED
:assemble SKIPPED
:compileTestKotlin SKIPPED
:compileTestJava SKIPPED
:processTestResources SKIPPED
:testClasses SKIPPED
:test SKIPPED
:check SKIPPED
:build SKIPPED

BUILD SUCCESSFUL in 0s
```

The Kotlin plug-in adds the `compileKotlin`, `inspectClassesForKotlinIC`, and `compileTestKotlin` tasks.

The project can be built by using the same command without the `-m` flag, which is the abbreviation for `--dry-run`.

1.10 Using Maven with Kotlin

Problem

You want to compile Kotlin by using the Maven build tool.

Solution

Use the Kotlin Maven plug-in and standard library dependencies.

Discussion

The basic details for using Maven can be found on the documentation web page.

This documentation recommends that first you specify the Kotlin version you want to use as a property in a Maven *pom.xml* file that looks like this:

```
    <properties>
        <kotlin.version>1.3.50</kotlin.version>
    </properties>
```

Then, add the Kotlin standard library as a dependency, as in Example 1-15.

Example 1-15. Adding the Kotlin standard library dependency

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

As with Gradle, you can specify `kotlin-stdlib-jdk7` or `kotlin-stdlib-jdk8` to use extension functions for Java 1.7 or 1.8.

Additional available artifact IDs include `kotlin-reflect` for reflection, and `kotlin-test` and `kotlin-test-junit` for testing.

To compile Kotlin source code, tell Maven in which directories it is located, as in Example 1-16.

Example 1-16. Specifying Kotlin source directories

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

Then tell the Kotlin plug-in to compile the sources and tests (Example 1-17).

Example 1-17. Referencing the Kotlin plug-in

```
<build>
    <plugins>
        <plugin>
            <artifactId>kotlin-maven-plugin</artifactId>
            <groupId>org.jetbrains.kotlin</groupId>
            <version>${kotlin.version}</version>

            <executions>
                <execution>
                    <id>compile</id>
                    <goals><goal>compile</goal></goals>
                </execution>

                <execution>
                    <id>test-compile</id>
                    <goals><goal>test-compile</goal></goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

When your project contains both Kotlin code and Java code, the Kotlin compiler should be invoked first. That means `kotlin-maven-plugin` should be run before `maven-compiler-plugin`. The documentation page provided previously shows how to ensure that via configuration options in your *pom.xml* file.

CHAPTER 2

Basic Kotlin

This chapter contains recipes that work with the fundamentals of Kotlin. They show you how to use the language without relying on specific libraries.

2.1 Using Nullable Types in Kotlin

Problem

You want to ensure that a variable is never null.

Solution

Define the type of a variable without a question mark. Nullable types also combine with the safe call operator (`?.`) and the Elvis operator (`?:`)

Discussion

The most attractive feature of Kotlin may be that it eliminates almost all possible nulls. In Kotlin, if you define a variable without including a trailing question mark, the compiler will require that value to be non-null, as in Example 2-1.

Example 2-1. Declaring a non-nullable variable

```
var name: String

// ... later ...
name = "Dolly" ❶
// name = null ❷
```

❶ Assignment to a non-null string

❷ Assignment to `null` does not compile

Declaring the `name` variable to be of type `String` means that it cannot be assigned the value `null` or the code won't compile.

If you do want a variable to allow `null`, add a question mark to the type declaration, as in Example 2-2.

Example 2-2. Declaring a nullable variable

```
class Person(val first: String,
             val middle: String?,
             val last: String)

val jkRowling = Person("Joanne", null, "Rowling") ❶
val northWest = Person("North", null, "West")     ❷
```

❶ JK Rowling has no given middle name; she selected K for her initial to honor her grandmother Katherine

❷ Neither does Kim and Kanye's baby, who will no doubt have bigger issues than this

In the `Person` class shown, you still have to supply a value for the `middle` parameter, even if it's null.

Life gets interesting when you try to use a nullable variable in an expression. Kotlin requires you to check that the value is not null, but it's not quite as easy as that sounds. For example, consider the null check in Example 2-3.

Example 2-3. Checking nullability of a `val` variable

```
val p = Person(first = "North", middle = null, last = "West")
if (p.middle != null) {
    val middleNameLength = p.middle.length  ❶
}
```

❶ Smart cast to non-null `String`

The `if` test checks whether the `middle` property is non-null, and if so, Kotlin performs a *smart cast*: it treats `p.middle` as though it was of type `String` rather than `String?`. This works, but only because the variable `p` was declared with the `val` keyword, so it cannot change once it is set. If, on the other hand, the variable was declared with `var` instead of `val`, the code is as shown in Example 2-4.

Example 2-4. Asserting that a `var` variable is not null

```
var p = Person(first = "North", middle = null, last = "West")
if (p.middle != null) {
    // val middleNameLength = p.middle.length    ❶
    val middleNameLength = p.middle!!.length     ❷
}
```

❶ Smart cast to `String` impossible, because `p.middle` is a complex expression

❷ Null-asserted (please don't do this unless absolutely necessary)

Because `p` uses `var` instead of `val`, Kotlin assumes that it could change between the time it is defined and the time the `middle` property is accessed, and refuses to do the smart cast. One way around this is to use the bang-bang, or not-null, assertion operator (`!!`) which is a code smell if ever there was one. The `!!` operator forces the variable to be treated as non-null and throws an exception if that is not correct. That's one of the few ways it is still possible to get a `NullPointerException` even in Kotlin code, so try to avoid it if possible.

A better way to handle this situation is to use the *safe call* operator (`?.`). Safe call short-circuits and returns a null if the value is `null`, as in Example 2-5.

Example 2-5. Using the safe call operator

```
var p = Person(first = "North", middle = null, last = "West")
val middleNameLength = p.middle?.length    ❶
```

❶ Safe call; the resulting type is `Int?`

The problem is that the resulting inferred type is also nullable, so `middleNameLength` is of type `Int?`, which is probably not what you want. Therefore, it is helpful to combine the safe call operator with the *Elvis operator* (`?:`), as in Example 2-6.

Example 2-6. Safe call with Elvis

```
var p = Person(first = "North", middle = null, last = "West")
val middleNameLength = p.middle?.length ?: 0    ❶
```

❶ Elvis operator, returns 0 if `middle` is null

The Elvis operator checks the value of the expression to the left, and if it is not null, returns it. Otherwise, the operator returns the value of the expression on the right. In this case, it checks the value of `p.middle?.length`, which is either an integer or null. If it is an integer, the value is returned. Otherwise, the expression returns 0.

The righthand side of an Elvis operator can be an expression, so you can use `return` or `throw` when checking function arguments.

The real challenge, perhaps, is looking at `?:`, turning your head to the side, and somehow managing to see Elvis Presley. Clearly, Kotlin was designed for developers with an active imagination.[1]

Finally, Kotlin provides a *safe cast* operator, `as?`. The idea is to avoid throwing a `ClassCastException` if the cast isn't going to work. For example, if you try to cast an instance of `Person` to that type, but the value may be null, you can write the code shown in Example 2-7.

Example 2-7. The safe cast operator

```
val p1 = p as? Person ❶
```

❶ Variable `p1` is of type `Person?`

The cast will either succeed, resulting in a `Person`, or will fail, and `p1` will receive `null` as a result.

2.2 Adding Nullability Indicators to Java

Problem

Your Kotlin code needs to interact with Java code, and you want it to enforce nullability annotations.

Solution

Enforce JSR-305 nullability annotations in your Kotlin code, using the compile-time parameter `-Xjsr305=strict`.

Discussion

One of Kotlin's primary features is that nullability is enforced in the type system at compile time. If you declare a variable to be of type `String`, it can never be null, whereas if it is declared to be of type `String?`, it can, as in Example 2-8.

1 Or, rather, Groovy was designed that way. The Elvis operator is borrowed from Groovy.

Example 2-8. Nullable versus non-nullable types

```
var s: String  = "Hello, World!"  ❶
var t: String? = null             ❷
```

❶ Cannot be null, or code won't compile

❷ Question mark on type indicates a nullable type

This is fine until you want to interact with Java code, which has no such mechanism built into the language. Java does, however, have a `@Nonnull` annotation defined in the *javax.annotation* package. While this specification is now considered dormant, many libraries have what are referred to as *JSR-305 compatible* annotations, and Kotlin supports them.

For example, when using the Spring Framework, you can enforce compatibility by adding the code in Example 2-9 to your Gradle build file.

Example 2-9. Enforcing JSR-305 compatibility in Gradle (Groovy DSL)

```
sourceCompatibility = 1.8
compileKotlin {
    kotlinOptions {
        jvmTarget = "1.8"
        freeCompilerArgs = ["-Xjsr305=strict"]
    }
}
compileTestKotlin {
    kotlinOptions {
        jvmTarget = "1.8"
        freeCompilerArgs = ["-Xjsr305=strict"]
    }
}
```

To do the same using Gradle's Kotlin DSL, see Example 2-10.

Example 2-10. Enforcing JSR-305 compatibility in Gradle (Kotlin DSL)

```
tasks.withType<KotlinCompile> {
    kotlinOptions {
        jvmTarget = "1.8"
        freeCompilerArgs = listOf("-Xjsr305=strict")
    }
}
```

For Maven, add the snippet from Example 2-11 to the POM file, as described in the Kotlin reference guide.

Example 2-11. Enforcing JSR-305 compatibility in Maven

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <executions>...</executions>
    <configuration>
        <nowarn>true</nowarn>  <!-- Disable warnings -->
        <args>
            <!-- Enable strict mode for JSR-305 annotations -->
            <arg>-Xjsr305=strict</arg>
            ...
        </args>
    </configuration>
</plugin>
```

The `@Nonnull` annotation defined in JSR-305 takes a property called `when`. If the value of `when` is `When.ALWAYS`, the annotated type is treated as non-null. If it is `When.MAYBE` or `When.NEVER`, it is considered nullable. If it is `When.UNKNOWN`, the type is assumed to be a *platform* type whose nullability is not known.

2.3 Adding Overloaded Methods for Java

Problem

You have a Kotlin function with default parameters, and you want to invoke it from Java without having to specify explicit values for each parameter.

Solution

Add the `@JvmOverloads` annotation to the function.

Discussion

Say you have a Kotlin function that specifies one or more default arguments, as in Example 2-12.

Example 2-12. A Kotlin function with default parameters

```
fun addProduct(name: String, price: Double = 0.0, desc: String? = null) =
    "Adding product with $name, ${desc ?: "None" }, and " +
            NumberFormat.getCurrencyInstance().format(price)
```

For the `addProduct` function, a `String` name is required, but the description and price have default values. The description is nullable and defaults to `null`, while the price defaults to 0.

It is easy enough to call this function from Kotlin with one, two, or three arguments, as the test in Example 2-13 shows.

Example 2-13. Calling the overloaded variations from Kotlin

```
@Test
fun `check all overloads`() {
    assertAll("Overloads called from Kotlin",
        { println(addProduct("Name", 5.0, "Desc")) },
        { println(addProduct("Name", 5.0)) },
        { println(addProduct("Name")) }
    )
}
```

Each call to `addProduct` uses one fewer argument than the previous one.

> Optional or nullable properties should be placed at the end of a function signature, so they can be left out when calling the function with positional arguments.

All of the calls execute properly.

Java, however, does not support default arguments for methods, so when calling this function from Java, you have to supply them all, as in Example 2-14.

Example 2-14. Calling the function from Java

```
@Test
void supplyAllArguments() {
    System.out.println(OverloadsKt.addProduct("Name", 5.0, "Desc"));
}
```

If you add the annotation `@JvmOverloads` to the function, the generated class will support all the function overloads, as in Example 2-15.

Example 2-15. Calling all the overloads from Java

```
@Test
void checkOverloads() {
    assertAll("overloads called from Java",
        () -> System.out.println(OverloadsKt.addProduct("Name", 5.0, "Desc")),
        () -> System.out.println(OverloadsKt.addProduct("Name", 5.0)),
        () -> System.out.println(OverloadsKt.addProduct("Name"))
    );
}
```

To see why this works, you can decompile the generated bytecodes from Kotlin. Without the `@JvmOverloads` annotation, the generated code resembles Example 2-16.

Example 2-16. Decompiled function from Kotlin bytecodes

```
@NotNull
public static final String addProduct(@NotNull String name,
    double price, @Nullable String desc) {
    Intrinsics.checkParameterIsNotNull(name, "name");
    // ...
}
```

When you add the `@JvmOverloads` annotation, the result instead resembles Example 2-17.

Example 2-17. Decompiled function with overloads

```
// public final class OverloadsKt {
@JvmOverloads
@NotNull
public static final String addProduct(@NotNull String name,
    double price, @Nullable String desc) {
    Intrinsics.checkParameterIsNotNull(name, "name");

   // ...
}

@JvmOverloads
@NotNull
public static final String addProduct(
    @NotNull String name, double price) {
    return addProduct$default(name, price,
        (String)null, 4, (Object)null);
}

@JvmOverloads
@NotNull
public static final String addProduct(@NotNull String name) {
    return addProduct$default(name, 0.0D,
        (String)null, 6, (Object)null);
}
```

The generated class includes additional methods that invoke the full method with supplied, default arguments.

You can also do this with constructors. The `Product` class shown in Example 2-18 generates three constructors: one with all three arguments, one with only the name and price, and one with only the name.

Example 2-18. Kotlin class with overloaded constructors

```
data class Product @JvmOverloads constructor(
    val name: String,
    val price: Double = 0.0,
    val desc: String? = null
)
```

The explicit `constructor` call is necessary so that you can add the `@JvmOverloads` annotation to it. Now, instantiating the class can be done with multiple arguments in Kotlin, as in Example 2-19.

Example 2-19. Instantiating the `Product` class from Kotlin

```
@Test
internal fun `check overloaded Product contructor`() {
    assertAll("Overloads called from Kotlin",
        { println(Product("Name", 5.0, "Desc")) },
        { println(Product("Name", 5.0)) },
        { println(Product("Name")) }
    )
}
```

Or you can call the constructors from Java, as in Example 2-20.

Example 2-20. Instantiating the `Product` class from Java

```
@Test
void checkOverloadedProductCtor() {
    assertAll("overloads called from Java",
        () -> System.out.println(new Product("Name", 5.0, "Desc")),
        () -> System.out.println(new Product("Name", 5.0)),
        () -> System.out.println(new Product("Name"))
    );
}
```

This all works, but note that a subtle trap exists. If you look at the decompiled code for the `Product` class, you'll see all the necessary constructors, shown in Example 2-21.

Example 2-21. Overloaded `Product` constructors in decompiled code

```
@JvmOverloads
public Product(@NotNull String name, double price,
    @Nullable String desc) {
    Intrinsics.checkParameterIsNotNull(name, "name");
    super();
    this.name = name;
    this.price = price;
    this.desc = desc;
}

@JvmOverloads
public Product(String var1, double var2, String var4,
    int var5, DefaultConstructorMarker var6) {

    // ...

    this(var1, var2, var4);  ❶
}

@JvmOverloads
public Product(@NotNull String name, double price) {
    this(name, price, (String)null, 4, (DefaultConstructorMarker)null); ❷
}

@JvmOverloads
public Product(@NotNull String name) {
   this(name, 0.0D, (String)null, 6, (DefaultConstructorMarker)null);   ❷
}
```

❶ Calls three-argument constructor

❷ Calls generated constructor, which then calls three-argument constructor

Each of the overloaded constructors ultimately calls the full, three-argument version with various defaults supplied. This is probably fine, but keep in mind that when you invoke a constructor with optional arguments, you're not calling the analogous constructor in the superclass; all calls are going through a single constructor with the most arguments.

> Calls to constructors marked `@JvmOverloads` do not call `super` with the same number of arguments. Instead, they call the full constructor with supplied defaults.

In Java, each constructor calls its parent by using `super`, and when you overload constructors, you often invoke `super` with the same number of arguments. That's not happening in this case. The superclass constructor that gets invoked is the one with all the parameters, with supplied defaults.

2.4 Converting Between Types Explicitly

Problem

Kotlin does not automatically promote primitive types to wider variables, such as an `Int` to a `Long`.

Solution

Use the explicit conversion functions `toInt`, `toLong`, and so on to convert the smaller type explicitly.

Discussion

One of the surprises Kotlin brings to Java developers is that shorter types are not automatically promoted to longer types. For example, in Java it is perfectly normal to write the code in Example 2-22.

Example 2-22. Promoting shorter primitive types to longer ones in Java

```
int myInt = 3;
long myLong = myInt;                        ❶
```

❶ Automatic promotion of `int` to `long`

When autoboxing was introduced in Java 1.5, it became easy to convert from a primitive to a wrapper type, but converting from one wrapper type to another still requires extra code, as in Example 2-23.

Example 2-23. Converting from an `Integer` to a `Long`

```
Integer myInteger = 3;
// Long myWrappedLong = myInteger;          ❶
Long myWrappedLong = myInteger.longValue(); ❷
myWrappedLong = Long.valueOf(myInteger);    ❸
```

❶ Does not compile

❷ Extracts a `long`, then wraps it

❸ Unwraps `int`, promotes to `long`, then wraps it

In other words, dealing with the wrapped types directly requires you to do the unboxing yourself. You can't simply assign an `Integer` instance to a `Long` without extracting the wrapped value first.

In Kotlin, primitives are not supported directly. The bytecodes may generate their equivalents, but when you are writing the code yourself, you need to keep in mind that you are dealing with classes rather than primitives.

Fortunately, Kotlin provides a set of conversion methods of the form `toInt`, `toLong`, and so on, as illustrated in Example 2-24.

Example 2-24. Promoting an `Int` to a `Long` in Kotlin

```
val intVar: Int = 3
// val longVar: Long = intVar          ❶
val longVar: Long = intVar.toLong()    ❷
```

❶ Does not compile

❷ Explicit type conversion

Since `intVar` and `longVar` are instances of classes in Kotlin, perhaps being unable to automatically assign an instance of `Int` to a variable of type `Long` is not too surprising. But it is easy to forget that, especially if you have a Java background.

The available conversion methods are as follows:

- `toByte: Byte`
- `toChar: Char`
- `toShort: Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`

Fortunately, Kotlin does take advantage of operator overloading to perform type conversions transparently, so the following code does not require an explicit conversion:

```
val longSum = 3L + intVar
```

The `plus` operator automatically converts the `intVar` value to a `long` and adds it to the `long` literal.

2.5 Printing to Different Bases

Problem

You want to print a number in a base other than base 10.

Solution

Use the extension function `toString(radix: Int)` for a valid radix.

Discussion

This recipe is for a special situation that arises infrequently. Still, it's interesting and may be useful if you deal with alternate numerical bases.

There's an old joke:

```
There are 10 kinds of people
Those who know binary, and those who don't
```

In Java, if you wanted to print a number in binary, you would use the static `Integer.toBinaryString` method or the static `Integer.toString(int, int)` method. The first argument is the value to convert, and the second argument is the desired base.

Kotlin, however, took the static method in Java and made it the extension function `toString(radix: Int)` on `Byte`, `Short`, `Int`, and `Long`. For example, to convert the number 42 to a binary string in Kotlin, you would write Example 2-25.

Example 2-25. Printing 42 in binary

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

In binary, the bit positions from left to right are 1, 2, 4, 8, 16, and so on. Because 42 is 2 + 8 + 32, those bit positions have 1s and the others have 0s.

The implementation of the `toString` method in `Int` is given by the following:

```
public actual inline fun Int.toString(radix: Int): String =
    java.lang.Integer.toString(this, checkRadix(radix))
```

The extension function on `Int` thus delegates to the corresponding static method in `java.lang.Integer`, after checking the radix in the second argument.

The `actual` keyword indicates a platform-specific implementation in multiplatform projects.

The `checkRadix` method verifies that the specified base is between `Character.MIN_RADIX` and `Character.MAX_RADIX` (again, this is for the Java implementation), and throws an `IllegalArgumentException` otherwise. The valid min and max values are to 2 and 36, respectively. Example 2-26 shows the output of printing the number 42 in all the valid radices.

Example 2-26. Printing 42 in all available radix values

```
(Character.MIN_RADIX..Character.MAX_RADIX).forEach { radix ->
    println("$radix: ${42.toString(radix)}")
}
```

The output (with some formatting) is as follows:

```
Radix  Value
 2:   101010
 3:     1120
 4:      222
 5:      132
 6:      110
 7:       60
 8:       52
 9:       46
10:       42
...
32:       1a
33:       19
34:       18
35:       17
36:       16
```

The number 42 is the Answer to the Ultimate Question of Life, the Universe, and Everything (at least according to Douglas Adams in his *Hitchhiker's Guide to the Galaxy* series).

Combining this capability with multiline strings gives a Kotlin version of a nice variation of the original joke, as in Example 2-27.

Example 2-27. Improving the binary joke

```
val joke = """
    Actually, there are ${3.toString(3)} kinds of people
    Those who know binary, those who don't,
    And those who didn't realize this is actually a ternary joke
""".trimIndent()
println(joke)
```

That code prints the following:

```
    Actually, there are 10 kinds of people
    Those who know binary, those who don't,
    And those who didn't realize this is actually a ternary joke
```

2.6 Raising a Number to a Power

Problem

You want to raise a number to a power but notice that Kotlin doesn't have a predefined exponentiation operator.

Solution

Define an `infix` function that delegates to the Kotlin extension function `pow` already added to `Int` and `Long`.

Discussion

Kotlin, like Java, does not have a built-in exponentiation operator. Java at least includes the static `pow` function in the `java.lang.Math` class, whose signature is as follows:

```
public static double Math.pow(double a, double b)
```

Since Java automatically promotes shorter primitive types to longer ones (for example, `int` to `double`), this is the only function required to do the job. In Kotlin, however, there are no primitives, and instances of classes like `Int` are not automatically promoted to `Long` or `Double`. This becomes annoying when you notice that the Kotlin standard library does define an extension function called `pow` on `Float` and `Double`, but that there is no corresponding `pow` function in `Int` or `Long`.

The signatures for the existing functions are as follows:

```
fun Double.pow(x: Double): Double
fun Float.pow(x: Float): Float
```

That means to raise an integer to a power, you need to go through a conversion to `Float` or `Double` first, then invoke `pow`, and finally convert the result back to the original type, as in Example 2-28.

Example 2-28. Raising an `Int` to a power

```
@Test
fun `raise an Int to a power`() {
    assertThat(256, equalTo(2.toDouble().pow(8).toInt()))
}
```

If all you want to do is raise to a power of 2, the `shl` and `shr` functions are ideal, as shown in Recipe 2.7.

That works, but the process can be automated by defining extension functions on `Int` and `Long` with the following signatures:

```
fun Int.pow(x: Int) = toDouble().pow(x).toInt()
fun Long.pow(x: Int) = toDouble().pow(x).toLong()
```

This might be better done as an `infix` operator. Although only the predefined operator symbols can be overloaded, you can fake one by enclosing it in backticks, as in Example 2-29.

Example 2-29. Defining an `infix` operation for exponentiation

```
import kotlin.math.pow

infix fun Int.`**`(x: Int) = toDouble().pow(x).toInt()
infix fun Long.`**`(x: Int) = toDouble().pow(x).toLong()
infix fun Float.`**`(x: Int) = pow(x)
infix fun Double.`**`(x: Int) = pow(x)

// Pattern similar to existing functions on Float and Double
fun Int.pow(x: Int) = `**`(x)
fun Long.pow(x: Int) = `**`(x)
```

The `infix` keyword was used in the definition of the `**` function, but not in extending `pow` to `Int` and `Long` to keep with the pattern in `Float` and `Double`.

The result is that you can use the `**` symbol as a synthesized exponentiation operator, as in Example 2-30.

*Example 2-30. Using the ** extension function*

```
@Test
fun `raise to power`() {
    assertAll(
        { assertThat(1, equalTo(2 `**` 0)) },
        { assertThat(2, equalTo(2 `**` 1)) },
        { assertThat(4, equalTo(2 `**` 2)) },
        { assertThat(8, equalTo(2 `**` 3)) },

        { assertThat(1L, equalTo(2L `**` 0)) },
        { assertThat(2L, equalTo(2L `**` 1)) },
        { assertThat(4L, equalTo(2L `**` 2)) },
        { assertThat(8L, equalTo(2L `**` 3)) },

        { assertThat(1F, equalTo(2F `**` 0)) },
        { assertThat(2F, equalTo(2F `**` 1)) },
        { assertThat(4F, equalTo(2F `**` 2)) },
        { assertThat(8F, equalTo(2F `**` 3)) },

        { assertThat(1.0, closeTo(2.0 `**` 0, 1e-6)) },
        { assertThat(2.0, closeTo(2.0 `**` 1, 1e-6)) },
        { assertThat(4.0, closeTo(2.0 `**` 2, 1e-6)) },
        { assertThat(8.0, closeTo(2.0 `**` 3, 1e-6)) },

        { assertThat(1, equalTo(2.pow(0))) },
        { assertThat(2, equalTo(2.pow(1))) },
        { assertThat(4, equalTo(2.pow(2))) },
        { assertThat(8, equalTo(2.pow(3))) },

        { assertThat(1L, equalTo(2L.pow(0))) },
        { assertThat(2L, equalTo(2L.pow(1))) },
        { assertThat(4L, equalTo(2L.pow(2))) },
        { assertThat(8L, equalTo(2L.pow(3))) }
    )
}
```

The tests on `Double.**` use the Hamcrest matcher `closeTo` to avoid comparing for equality on doubles. The set of tests with `Float` probably should do the same, but the tests currently pass as they are.

The idea of defining an `infix` function for this purpose was suggested by an answer by Olivia Zoe to a question on Stack Overflow.

If you find wrapping the star-star operator in backticks annoying, it's easy enough to define an actual function name, like `exp`, instead.

2.7 Using Bitwise Shift Operators

Problem

You want to perform bitwise shift operations.

Solution

Kotlin includes bitwise infix functions like `shr`, `shl`, and `ushr` for this purpose.

Discussion

Bitwise operations come up in a variety of applications, including access control lists, communication protocols, compression and encryption algorithms, and computer graphics. Unlike many other languages, Kotlin does not use specific operator symbols for shifting operations, but instead defines functions for them.

Kotlin defines the following shift operators as extension functions on `Int` and `Long`:

`shl`
: Signed left shift

`shr`
: Signed right shift

`ushr`
: Unsigned right shift

Because of two's complement arithmetic, shifting bits left or right is like multiplying or dividing by 2, as shown in Example 2-31.

Example 2-31. Multiplying and dividing by 2

```
@Test
fun `doubling and halving`() {
    assertAll("left shifts doubling from 1",    // 0000_0001
        { assertThat(  2, equalTo(1 shl 1)) },  // 0000_0010
        { assertThat(  4, equalTo(1 shl 2)) },  // 0000_0100
        { assertThat(  8, equalTo(1 shl 3)) },  // 0000_1000
        { assertThat( 16, equalTo(1 shl 4)) },  // 0001_0000
        { assertThat( 32, equalTo(1 shl 5)) },  // 0010_0000
        { assertThat( 64, equalTo(1 shl 6)) },  // 0100_0000
        { assertThat(128, equalTo(1 shl 7)) }   // 1000_0000
    )

    assertAll("right shifts halving from 235",   // 1110_1011
        { assertThat(117, equalTo(235 shr 1)) }, // 0111_0101
        { assertThat( 58, equalTo(235 shr 2)) }, // 0011_1010
        { assertThat( 29, equalTo(235 shr 3)) }, // 0001_1101
        { assertThat( 14, equalTo(235 shr 4)) }, // 0000_1110
        { assertThat(  7, equalTo(235 shr 5)) }, // 0000_0111
        { assertThat(  3, equalTo(235 shr 6)) }, // 0000_0011
        { assertThat(  1, equalTo(235 shr 7)) }  // 0000_0001
    )
}
```

The `ushr` function is needed when you want to shift a value and not preserve its sign. Both `shr` and `ushr` behave the same for positive values. But for negative values, `shr` fills in from the left with 1s so that the resulting value is still negative, as shown in Example 2-32.

Example 2-32. Using the `ushr` function versus `shr`

```
val n1 = 5
val n2 = -5
println(n1.toString(2))  //  0b0101
println(n2.toString(2))  // -0b0101

assertThat(n1  shr 1, equalTo(0b0010))      // 2
assertThat(n1 ushr 1, equalTo(0b0010))      // 2

assertThat(n2  shr 1, equalTo(-0b0011))     // -3
assertThat(n2 ushr 1, equalTo(0x7fff_fffd)) // 2_147_483_645
```

The seemingly strange behavior of the last example comes from two's complement arithmetic. Because `ushr` fills in from the left with 0s, it does not preserve the negative sign of `-3`. The result is the two's complement of –3 for a 32-bit integer, giving the value shown.

The `ushr` function comes up in many places. One interesting example occurs when trying to find the midpoint of two large integers, as in Example 2-33.

Example 2-33. Finding the midpoint of two large integers

```
val high = (0.99 * Int.MAX_VALUE).toInt()
val low  = (0.75 * Int.MAX_VALUE).toInt()

val mid1 = (high + low) / 2     ❶
val mid2 = (high + low) ushr 1  ❷

assertTrue(mid1 !in low..high)
assertTrue(mid2  in low..high)
```

❶ Sum is greater than max `Int`, so result is negative

❷ Unsigned shift ensures result inside desired range

If both values are large, adding them together will produce a result larger than `Int.MAX_VALUE`, so the sum will be negative. By doing an unsigned right shift to divide by 2, the result is between the low and high values.

Many algorithms, like binary searches or sorts, require computing the mean of two integers, each of which could potentially be very large. Using `ushr` in this way ensures that the result is bounded in the way you want.

2.8 Using Bitwise Boolean Operators

Problem

You want to apply masks to bit values.

Solution

Use the bitwise `and`, `or`, `xor`, and `inv` operators supplied by Kotlin for that purpose.

Discussion

In addition to the shift operators defined on `Int` and `Long`, Kotlin defines masking operations `and`, `or`, `xor`, and `inv` (rather than "not").

Taking the last one first, the `inv` function flips all the bits on a number. As a simple example, the number 4 in binary is `0b00000100`. Flipping all the bits gives `0b11111011`, which is 251 in decimal. When you invoke the `inv` function on 4, however, you get –5, as shown in Example 2-34.

Precede a numeric literal with `0b` to express it in binary.

Example 2-34. Inverse of 4

```
// 4 == 0b0000_0100 (in binary)
// Bitwise complement (flipping all the bits) is given by:
//      0b1111_1011 == 251 (in decimal)
assertEquals(-5, 4.inv())
```

You can add underscores (`_`) to numeric literals to make them easier to read. They are ignored by the compiler.

Why do you get –5 instead of 251? The system is doing two's complement arithmetic. For any integer `n`, the two's complement of `n` is given by `-(~n + 1)`, where `~n` is the one's complement (i.e., flip all the bits) of `n`. Therefore:

```
0b1111_1011 -> -(0b0000_0100 + 1) -> -0b0000_0101 -> -5
```

Hence the two's complement of 251 is –5.

The bitwise operations `and`, `or`, and `xor` are familiar to most developers. The only difference between them and their logical counterparts is that they do not short-circuit. As a trivial example, see Example 2-35.

Example 2-35. Simple example of `and`, `or`, and `xor`

```
@Test
fun `and, or, xor`() {
    val n1 = 0b0000_1100       // decimal 12
    val n2 = 0b0001_1001       // decimal 25

    val n1_and_n2 = n1 and n2
    val n1_or_n2  = n1 or n2
    val n1_xor_n2 = n1 xor n2

    assertThat(n1_and_n2, equalTo(0b0000_1000)) //  8
    assertThat(n1_or_n2,  equalTo(0b0001_1101)) // 29
    assertThat(n1_xor_n2, equalTo(0b0001_0101)) // 21
}
```

For a more interesting example, consider the RGBA model for representing colors, as exemplified by the `java.awt.Color` class in Java. A color can be represented as a 4-byte integer, where 1 byte contains the values for red, green, blue, and alpha (a measure of transparency). See Figure 2-1 for details.

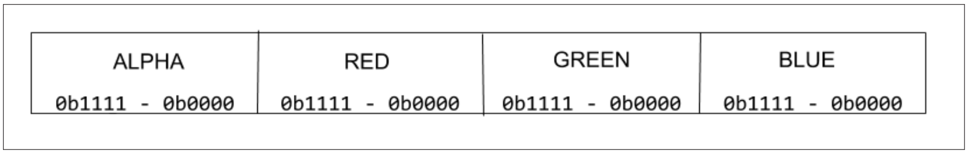


Figure 2-1. 32-bit integer with 1 byte for each color

Given an instance of the `Color` class, the `getRGB` method documentation says that the method returns an `int` for "the RGB value representing the color in the default sRGB ColorModel (bits 24–31 are alpha, 16–23 are red, 8–15 are green, 0–7 are blue)."

That means that given the returned integer, Kotlin can extract the actual RGB and alpha values by using a function like that in Example 2-36.

Example 2-36. Converting an integer to its individual RGB values

```
fun intsFromColor(color: Color): List<Int> {
    val rgb   = color.rgb                  ❶
    val alpha = rgb shr 24 and 0xff        ❷
    val red   = rgb shr 16 and 0xff        ❷
    val green = rgb shr  8 and 0xff        ❷
    val blue  = rgb and        0xff        ❷
    return listOf(alpha, red, green, blue)
}
```

❶ Invokes Java's `getRGB` method

❷ Shifts right and applies mask to return the proper `Int`

The advantage to returning the individual values in a list is that you can use destructuring in a test, as in Example 2-37.

Example 2-37. Destructuring and testing

```
@Test
fun `colors as ints`() {
    val color = Color.MAGENTA
    val (a, r, g, b) = intsFromColor(color)

    assertThat(color.alpha, equalTo(a))
    assertThat(color.red,   equalTo(r))
    assertThat(color.green, equalTo(g))
    assertThat(color.blue,  equalTo(b))
}
```

Going in the other direction, from the RGB values as integers, you can build up the overall `Int` value as shown in Example 2-38.

Example 2-38. Creating an `Int` from individual RGB and alpha values

```
fun colorFromInts(alpha: Int, red: Int, green: Int, blue: Int) =
    (alpha and 0xff shl 24) or
    (red   and 0xff shl 16) or
    (green and 0xff shl 8) or
    (blue  and 0xff)
```

This time, the values are shifted left rather than right. That's easy enough to test as well, as shown in Example 2-39.

Example 2-39. Converting from RGB and alpha to an `Int`

```
@Test
fun `ints as colors`() {
    val color = Color.MAGENTA
    val intColor = colorFromInts(color.alpha,
        color.red, color.green, color.blue)
    val color1 = Color(intColor, true)  ❶
    assertThat(color1, equalTo(color))
}
```

❶ Second constructor arg indicates alpha value is present

To end this recipe, consider the following `xor` joke: "An xor-cist eliminates one daemon or the other, but not both." Sorry about that, but feel free to inflict that joke on your friends.

2.9 Creating Pair Instances with to

Problem

You want to create instances of the `Pair` class (often as entries for a map).

Solution

Rather than instantiate the `Pair` class directly, use the infix `to` function.

Discussion

Maps are made up of entries, which are combinations of keys and values. To create a map, Kotlin provides a handful of top-level functions, like `mapOf`, that allow you to create a map from a list of `Pair` instances. The signature of the `mapOf` function in the standard library is as follows:

```
fun <K, V> mapOf(vararg pairs: Pair<K, V>): Map<K, V>
```

`Pair` is a data class that holds two elements, called `first` and `second`. The signature of the `Pair` class is shown here:

```
data class Pair<out A, out B> : Serializable
```

The `Pair` class properties `first` and `second` correspond to the generic values of `A` and `B`.

Although you can create a `Pair` class by using the two-argument constructor, it is more common to use the `to` function. The `to` function is defined as follows:

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

The implementation of the `to` function is to instantiate the `Pair` class.

Putting all of these features together, Example 2-40 shows how to create a map with pairs supplied by the `to` function.

Example 2-40. Using the `to` function to create pairs for `mapOf`

```
@Test
fun `create map using infix to function`() {
    val map = mapOf("a" to 1, "b" to 2, "c" to 2) ❶
    assertAll(
        { assertThat(map, hasKey("a")) },
        { assertThat(map, hasKey("b")) },
        { assertThat(map, hasKey("c")) },
        { assertThat(map, hasValue(1)) },
        { assertThat(map, hasValue(2)) })
}

@Test
fun `create a Pair from constructor vs to function`() {
    val p1 = Pair("a", 1)                              ❷
    val p2 = "a" to 1                                  ❶

    assertAll(
        { assertThat(p1.first, `is`("a")) },
        { assertThat(p1.second, `is`(1)) },
        { assertThat(p2.first, `is`("a")) },
        { assertThat(p2.second, `is`(1)) },
        { assertThat(p1, `is`(equalTo(p2)))}
    )
}
```

❶ Creates `Pair` using `to`

❷ Creates `Pair` from constructor

The `to` function is an extension function added to any generic type `A`, with generic argument `B`, that returns an instance of `Pair` combining the values supplied for `A` and `B`. It is simply a way to create map literals with less noise.

Incidentally, because `Pair` is a data class, the individual elements can be accessed by using destructuring, as in Example 2-41.

Example 2-41. Destructuring `Pair`

```
@Test
fun `destructuring a Pair`() {
    val pair = "a" to 1
    val (x,y) = pair

    assertThat(x, `is`("a"))
    assertThat(y, `is`(1))
}
```

There is also a class in the standard library called `Triple` that represents a triad of values. There are no convenient extension functions for creating `Triple` instances, however; you use the three-argument constructor directly.

CHAPTER 3

Object-Oriented Programming in Kotlin

Like Java, Kotlin is an object-oriented programming (OOP) language. As such, it uses classes, both abstract and concrete, and interfaces in a way that is familiar to Java developers.

Some aspects of OOP in Kotlin are worth spending additional time on, and this chapter does so. It includes recipes that involve initializing objects, providing custom getters and setters, performing late and lazy initialization, creating singletons, understanding the `Nothing` class, and more.

3.1 Understanding the Difference Between const and val

Problem

You need to indicate that a value is a compile-time rather than a runtime constant.

Solution

Use the modifier `const` for compile-time constants. The keyword `val` indicates that a variable cannot be changed once it is assigned, but that assignment can occur at runtime.

Discussion

The Kotlin keyword `val` indicates a variable that cannot be changed. In Java, the keyword `final` is used for the same purpose. Given that, why does Kotlin also support the modifier `const`?

Compile-time constants must be top-level properties or members of an object declaration or a companion object. They must be of type `String` or a primitive type wrapper class (Byte, Short, Int, Long, Float, Double, Char, or Boolean), and they cannot have a custom getter function. They must be assigned outside any function, including main, because their values must be known at compile time.

As an example, consider defining a min and max priority for a task, as in Example 3-1.

Example 3-1. Defining compile-time constants

```
class Task(val name: String, _priority: Int = DEFAULT_PRIORITY) {

    companion object {
        const val MIN_PRIORITY = 1                  ❶
        const val MAX_PRIORITY = 5                  ❶
        const val DEFAULT_PRIORITY = 3              ❶
    }

    var priority = validPriority(_priority)         ❷
        set(value) {
            field = validPriority(value)
        }

    private fun validPriority(p: Int) =             ❸
        p.coerceIn(MIN_PRIORITY, MAX_PRIORITY)
}
```

❶ Compile-time constants

❷ Property with custom setter

❸ Private validation function

In this example, three constants are defined using the normal Kotlin (and Java) idiom that suggests writing them in all uppercase letters. This example also takes advantage of a custom setter operation to map any provided priority into the given range.

Note that val is a Kotlin keyword, but const is a modifier, like private, inline, and so on. That's why const must be used along with the keyword val rather than replacing it.

See also

Custom setter methods like the one shown in this recipe are covered in Recipe 3.2.

3.2 Creating Custom Getters and Setters

Problem

You want to customize how a value is processed when assigned or returned.

Solution

Add `get` and `set` functions to properties in a Kotlin class.

Discussion

As in other object-oriented languages, Kotlin classes combine data with functions that operate on that data, in a technique commonly known as *encapsulation*. Kotlin is unusual in that everything is public by default, because this seems to violate the principle of data hiding, wherein the data structure associated with information is assumed to be an implementation detail.

Kotlin resolves this dilemma in an unusual way: fields cannot be declared directly in Kotlin classes. That sounds strange when you can define properties in a class that look just like fields, as in Example 3-2.

Example 3-2. A class presenting a task

```
class Task(val name: String) {
    var priority = 3

    // ...
}
```

The `Task` class defines two properties, `name` and `priority`. One is declared in the primary constructor, while the other is a top-level member of the class. Both could have been defined in the constructor, of course, but this shows that you can use the alternative syntax shown. The downside to declaring `priority` in this way is that you won't be able to assign it when instantiating the class, though you could still use an `apply` block:

```
    var myTask = Task().apply { priority = 4 }
```

The advantage to defining a property in this way is that you can easily add a custom getter and setter. The full syntax for defining a property is shown here:

```
    var <propertyName>[: <PropertyType>] [= <property_initializer]
        [<getter>]
        [<setter>]
```

The initializer, getter, and setter are optional. The type is optional if it can be inferred from the initialized value or the getter return type, though this is not true of properties declared in the constructor.

Properties declared in constructors must include a type, even when they are assigned default values.

Example 3-3 shows a custom getter being used to compute `isLowPriority`.

Example 3-3. A custom getter for a derived property

```
val isLowPriority
    get() = priority < 3
```

As stated, the type of `isLowPriority` is inferred from the return type of the `get` function, which in this case is a boolean.

A custom setter is used every time a value is assigned to a property. To make sure that `priority` is between 1 and 5, for example, a custom setter can be used as in Example 3-4.

Example 3-4. A custom setter for `priority`

```
var priority = 3
    set(value) {
        field = value.coerceIn(1..5)
    }
```

Here, at last, we see the resolution of the public property / private field dilemma listed previously. Normally, when a property needs a backing field, Kotlin provides it automatically. Here, however, in the custom setter, the `field` identifier is used to reference the generated backing field. The `field` identifier can be used only in a custom getter or setter.

A backing field is generated for a property if it uses the default generated getter or setter or if a custom getter or setter references it through the `field` property. That means that the derived property `lowPriority` will not have one.

In the literature, the terms *getters* and *setters* are formally referred to as *accessors* and *mutators*, presumably because it's hard to charge large consulting fees for *get* and *set*.

To complete this example, imagine that you want to be able to assign a priority by using a constructor. One way to do that is to introduce a constructor parameter that *isn't* a property, by leaving out the var or val keyword. This then leads to the implementation of Task, shown in Example 3-1 and repeated here for reference:

```
class Task(val name: String, _priority: Int = DEFAULT_PRIORITY) {

    companion object {
        const val MIN_PRIORITY = 1
        const val MAX_PRIORITY = 5
        const val DEFAULT_PRIORITY = 3
    }

    var priority = validPriority(_priority)
        set(value) {
            field = validPriority(value)
        }

    private fun validPriority(p: Int) =
        p.coerceIn(MIN_PRIORITY, MAX_PRIORITY)
}
```

The parameter _priority is not a property, but rather just an argument to the constructor. It is used to initialize the actual priority property, and the custom setter is evaluated to coerce it into the desired range every time it changes. Note the term value here is just a dummy name; you can change it to anything you like, as with any function parameter.

See Also

The constants used in the Task example are discussed in Recipe 3.1.

3.3 Defining Data Classes

Problem

You want to create a class representing an entity, complete with implementations of equals, hashCode, toString, and more.

Solution

Use the keyword data when defining your class.

Discussion

Kotlin provides the keyword `data` to indicate that the purpose of a particular class is to hold data. In Java, when such a class represents information from a database table, it is known as an *entity*, and the concept of a data class is similar.

Adding the word `data` to a class definition causes the compiler to generate a whole series of functions, including consistent `equals` and `hashCode` functions, a `toString` function that shows the class and the property values, a `copy` function, and component functions used for destructuring.

For example, consider the `Product` class:

```
data class Product(
    val name: String,
    var price: Double,
    var onSale: Boolean = false
)
```

The compiler generates `equals` and `hashCode` functions based on the properties declared in the primary constructor. The algorithm used is the same one described by Joshua Bloch years ago in *Effective Java* (Addison-Wesley Professional). The tests in Example 3-5 show that they work properly.

Example 3-5. Using the generated `equals` and `hashCode` implementations

```
@Test
fun `check equivalence`() {
    val p1 = Product("baseball", 10.0)
    val p2 = Product("baseball", 10.0, false)

    assertEquals(p1, p2)
    assertEquals(p1.hashCode(), p2.hashCode())
}

@Test
fun `create set to check equals and hashcode`() {
    val p1 = Product("baseball", 10.0)
    val p2 = Product(price = 10.0, onSale = false, name = "baseball")

    val products = setOf(p1, p2)
    assertEquals(1, products.size)  ❶
}
```

❶ Duplicate not added

Since `p1` and `p2` are equivalent, when both are included in the `setOf` function, only one is added to the actual result.

A `toString` implementation converts the product into a string:

```
Product(name=baseball, price=10.0, onSale=false)
```

The `copy` method is an instance method that creates a new object that starts with the property values from the original and modifies only the supplied values, as the test in Example 3-6 shows.

Example 3-6. Testing the `copy` function

```
@Test
fun `change price using copy`() {
    val p1 = Product("baseball", 10.0)
    val p2 = p1.copy(price = 12.0)          ❶
    assertAll(
        { assertEquals("baseball", p2.name) },
        { assertThat(p2.price, `is`(closeTo(12.0, 0.01))) },
        { assertFalse(p2.onSale) }
    )
}
```

❶ Changes only the price

The test verifies that using `copy` with the `price` parameter changes only that value. Note that the Hamcrest matcher `closeTo` is used to compare prices, because using equality checks with floating-point values is not considered a good idea.

Note that the `copy` function performs only a shallow copy, not a deep one. To demonstrate this, consider an additional data class called `OrderItem`, as shown in Example 3-7.

Example 3-7. A class containing a `Product`

```
data class OrderItem(val product: Product, val quantity: Int)
```

The test shown in Example 3-8 instantiates an `OrderItem` and then makes a copy by using the `copy` function.

Example 3-8. Test demonstrating shallow copy

```
@Test
fun `data copy function is shallow`() {
    val item1 = OrderItem(Product("baseball", 10.0), 5)
    val item2 = item1.copy()

    assertAll(
        { assertTrue(item1 == item2) },
        { assertFalse(item1 === item2) },          ❶
        { assertTrue(item1.product == item2.product) },
        { assertTrue(item1.product === item2.product) }  ❷
    )
}
```

❶ `OrderItem` produced by `copy` is a different object

❷ `Product` inside both `OrderItem` instances is the same object

The test shows that although the two `OrderItem` instances are equivalent (by the `equals` function invoked via ==), they are still two separate objects because the referential equality operator === returns `false`. They both, however, share the same internal `Product` instance, because === on both contained references returns `true`.

Invoking `copy` on a data class performs a shallow copy, not a deep one.

In addition to the `copy` function, data classes add functions called `component1`, `component2`, and so on, that return the values of the properties. These functions are used for destructuring, as shown in the test in Example 3-9.

Example 3-9. Destructuring a `Product` instance

```
@Test
fun `destructure using component functions`() {
    val p = Product("baseball", 10.0)

    val (name, price, sale) = p        ❶
    assertAll(
        { assertEquals(p.name, name) },
        { assertThat(p.price, `is`(closeTo(price, 0.01))) },
        { assertFalse(sale) }
    )
}
```

❶ Destructures the product

You are free to override any of these functions (`equals`, `hashCode`, `toString`, `copy`, or any of the `_componentN_` functions) if you wish. You can add other functions as well.

If you don't want a property to be included in the generated functions, add the property to the class body rather than the primary constructor.

Data classes are a convenient way of representing classes whose primary purpose is to hold data. The standard library includes two data classes, `Pair` and `Triple`, for holding two or three properties of any generic types. If you need more than that, create your own data class.

3.4 The Backing Property Technique

Problem

You have a property of a class that you want to expose to clients, but you need to control how it is initialized or read.

Solution

Define a second property of the same type and use a custom getter and/or setter to provide access to the property you care about.

Discussion

Say you have a class called `Customer` and you want to keep a list of messages or notes you've saved regarding them. You don't necessarily want to load all the messages whenever you create an instance, however, so you create the class shown in Example 3-10.

Example 3-10. `Customer` class, version 1

```
class Customer(val name: String) {
    private var _messages: List<String>? = null          ❶

    val messages: List<String>                           ❷
        get() {                                          ❸
            if (_messages == null) {
                _messages = loadMessages()
            }
            return _messages!!
        }

    private fun loadMessages(): MutableList<String> =
        mutableListOf(
            "Initial contact",
            "Convinced them to use Kotlin",
            "Sold training class. Sweet."
        ).also { println("Loaded messages") }
}
```

❶ Nullable private property used for initialization

❷ Property to be loaded

❸ Private function

In this class, the property `messages` will hold the list of messages about that client. To avoid initializing it immediately, the additional property `_messages` is added, which is of the same type but nullable. The custom getter is used to check whether the messages have been loaded yet, and if not, loads them. The test in Example 3-11 accesses the messages.

Example 3-11. Accessing the messages in the customer

```
@Test
fun `load messages`() {
    val customer = Customer("Fred").apply { messages }   ❶
    assertEquals(3, customer.messages.size)              ❷
}
```

❶ Loads messages the first time

❷ Accesses the messages again, but already loaded

You can't load the messages by using a constructor property, because `_messages` is private. If you want the messages right away, as shown here, use the `apply` function. In this test, that invokes the getter method, which both loads the messages and prints the info message. The second time the property is accessed, the messages have already been loaded, and no print is seen.

While this is a useful illustration, it implements lazy loading the hard way. Much easier is the code in Example 3-12, which uses the built-in `lazy` delegate function.

Example 3-12. Lazy loading the messages by using `lazy`

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

    val messages: List<String> by lazy { loadMessages() }   ❶

    private fun loadMessages(): MutableList<String> =
        mutableListOf(
            "Initial contact",
            "Convinced them to use Kotlin",
            "Sold training class. Sweet."
        ).also { println("Loaded messages") }
}
```

❶ Uses the `lazy` delegate

Still, using a private backing field to enforce initialization of a property is a useful technique.

A variation on this is to provide a constructor argument to set a value but still enforce constraints on the property, as done in Example 3-1 and repeated here for simplicity:

```
class Task(val name: String, _priority: Int = DEFAULT_PRIORITY) {

    companion object {
        const val MIN_PRIORITY = 1
        const val MAX_PRIORITY = 5
        const val DEFAULT_PRIORITY = 3
    }

    var priority = validPriority(_priority)
        set(value) {
            field = validPriority(value)
        }

    private fun validPriority(p: Int) =
        p.coerceIn(MIN_PRIORITY, MAX_PRIORITY)
}
```

Note that the `_priority` property is not marked with `val`, indicating it is only a constructor argument rather than an actual property of the class. The property you care about, `priority`, has a custom setter to assign its value based on the constructor argument.

The backing property technique shows up fairly often in Kotlin classes, so it's worth understanding how it works.

See Also

The `lazy` delegate is discussed further in Recipe 8.2.

3.5 Overloading Operators

Problem

You want a client to be able to use operators such as + and * with classes defined in a library.

Solution

Use Kotlin's operator-overloading mechanism to implement the associated functions.

Discussion

Many operators, including addition, subtraction, and multiplication, are implemented in Kotlin as functions. When you use the +, -, or * symbols, you are delegating to those functions. That means by supplying those functions, you allow a client to use operators.

The classic example, given in the reference docs, is to provide a member function `unaryMinus` for a `Point` class, as in Example 3-13.

Example 3-13. Overriding the `unaryMinus` operator on `Point` (from reference docs)

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
   println(-point)  // prints "Point(x=-10, y=-20)"
}
```

The `operator` keyword is necessary when overriding all operator functions other than `equals`.

What if you want to add the relevant functions to a class that you didn't write? You can use extension functions to do the job.

For example, consider the `Complex` class in the (Java) library Apache Commons Math, which represents a complex number (one with real and imaginary parts). If you browse the Javadocs, you'll see that the class includes methods like `add`, `subtract`, and `multiply`. In Kotlin, the +, -, and * operators correspond to the functions `plus`, `minus`, and `times`. If you add extension functions to `Complex` to delegate to the existing functions, as in Example 3-14, you can then use the operators instead.

Example 3-14. Extension functions on `Complex`

```
import org.apache.commons.math3.complex.Complex

operator fun Complex.plus(c: Complex) = this.add(c)
operator fun Complex.plus(d: Double) = this.add(d)
operator fun Complex.minus(c: Complex) = this.subtract(c)
operator fun Complex.minus(d: Double) = this.subtract(d)
operator fun Complex.div(c: Complex) = this.divide(c)
operator fun Complex.div(d: Double) = this.divide(d)
operator fun Complex.times(c: Complex) = this.multiply(c)
operator fun Complex.times(d: Double) = this.multiply(d)
operator fun Complex.times(i: Int) = this.multiply(i)
operator fun Double.times(c: Complex) = c.multiply(this)
operator fun Complex.unaryMinus() = this.negate()
```

In each case, the extension function delegates to the existing method in the Java class. The test in Example 3-15 illustrates how to use the delegated operator functions.

Example 3-15. Using the operators with `Complex` instances

```
import org.apache.commons.math3.complex.Complex
import org.apache.commons.math3.complex.Complex.*  ❶

import org.hamcrest.MatcherAssert.assertThat
import org.hamcrest.Matchers.`is`
import org.hamcrest.Matchers.closeTo
import org.junit.jupiter.api.Test

import org.junit.jupiter.api.Assertions.*
import java.lang.Math.*                            ❷

internal class ComplexOverloadOperatorsKtTest {
    private val first = Complex(1.0, 3.0)
    private val second = Complex(2.0, 5.0)

    @Test
    internal fun plus() {
        val sum = first + second
        assertThat(sum, `is`(Complex(3.0, 8.0)))
    }

    @Test
    internal fun minus() {
        val diff = second - first
        assertThat(diff, `is`(Complex(1.0, 2.0)))
    }
    @Test
    internal fun negate() {
        val minus1 = -ONE ❶

        assertThat(minus1.real, closeTo(-1.0, 0.000001))
        assertThat(minus1.imaginary, closeTo(0.0, 0.000001))
    }

    @Test
    internal fun `Euler's formula`() {
        val iPI = I * PI  ❷

        assertTrue(Complex.equals(iPI.exp(), -ONE, 0.000001))
    }
}
```

❶ Import of `Complex.*` allows `ONE` instead of `Complex.ONE`

❷ Can use `I` and `PI` for `Complex.I` and `Math.PI`

In the last test, the `exp` function from `Complex` returns the value of `e^{arg}`, so the test demonstrates Euler's formula, `e^{i * PI} == -1`.

The tests illustrate many of the overloaded operators. If you are writing in Kotlin and using the `Complex` class, a little bit of operator overloading with extension functions lets you use the same operators you've used with regular numbers.

3.6 Using lateinit for Delayed Initialization

Problem

You don't have enough information to initialize a property in a constructor, but you don't want to have to make the property nullable as a result.

Solution

Use the `lateinit` modifier on your property.

Discussion

Use this technique sparingly, only when necessary. Cases such as the dependency injection described here are useful, but in general, consider alternatives like the lazy evaluation in Recipe 8.2 where possible.

Properties of a class that are declared as non-null are supposed to be initialized in a constructor. Sometimes, however, you don't have enough information at that time to give the property a value. This occurs in dependency injection frameworks, in which the injection doesn't happen until after all objects have been constructed, or in setup methods in unit tests. For such cases, use the `lateinit` modifier on the property.

For example, the Spring framework uses the annotation `@Autowired` to assign values to dependencies from the so-called application context. Again, since the value is set after the instances have already been created, mark it as `lateinit`, as in the test case shown in Example 3-16.

Example 3-16. Testing a Spring controller

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class OfficerControllerTests {
    @Autowired
    lateinit var client: WebTestClient                  ❶

    @Autowired
    lateinit var repository: OfficerRepository          ❶

    @Before
    fun setUp() {
        repository.addTestData()                        ❷
    }

    @Test
    fun `GET to route returns all officers in db`() {
        client.get().uri("/route")                      ❸
        // ... get data and check values ...
    }

    // ... other tests ...
}
```

❶ Initialized by autowiring

❷ Uses `repository` in the `setup` function

❸ Uses `client` in tests

The `lateinit` modifier can be used only on `var` properties declared inside the body of a class, and only when the property does not have a custom getter or setter. Since Kotlin 1.2, you can also use `lateinit` on top-level properties and even local variables. The type must be non-null, and it cannot be a primitive type.

By adding `lateinit`, you are promising to initialize the variable before it is first used. Otherwise, it throws an exception, as shown in Example 3-17.

Example 3-17. Behavior of `lateinit` properties

```kotlin
class LateInitDemo {
    lateinit var name: String
}

class LateInitDemoTest {
    @Test
    fun `unitialized lateinit property throws exception`() {
        assertThrows<UninitializedPropertyAccessException> {
            LateInitDemo().name
        }
    }

    @Test
    fun `set the lateinit property and no exception is thrown`() {
        assertDoesNotThrow { LateInitDemo().apply { name = "Dolly" } }
    }
}
```

Accessing the `name` property before it has been initialized throws an `UninitializedPropertyAccessException`, as the test shows.

Inside the class, you can check whether one of its properties has been initialized by using `isInitialized` on the property reference, as in Example 3-18.

Example 3-18. Using `isInitialized` on a property reference

```kotlin
class LateInitDemo {
    lateinit var name: String

    fun initializeName() {
        println("Before assignment: ${::name.isInitialized}")
        name = "World"
        println("After assignment: ${::name.isInitialized}")
    }
}

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

The output from executing the `initializeName` function is as follows:

```
Before assignment: false
After assignment: true
```

lateinit Versus lazy

The `lateinit` modifier is used on `var` properties with the restrictions discussed in this recipe. The `lazy` delegate takes a lambda that is evaluated the first time a property is accessed.

Use `lazy` if initialization is expensive and might never be done. Also, `lazy` can be used only for `val` properties, whereas `lateinit` can be applied only to `var` properties. Finally, `lateinit` properties can be initialized from anywhere the property is visible, so, as in one of the preceding examples, from outside the object.

See Also

The `lazy` delegate is discussed in Recipe 8.2.

3.7 Using Safe Casting, Reference Equality, and Elvis to Override equals

Problem

You want to provide a good implementation of the `equals` method in a class, so that instances can be checked for equivalence.

Solution

Use the reference equality operator (===), the safe casting function (`as?`), and the Elvis operator (`?:`) together.

Discussion

All object-oriented languages have the concept of object equivalence versus object equality. In Java, the double equals operator (==), is used to check whether two references are assigned to the same object. By contrast, the `equals` method, as part of the `Object` class, is intended to be overridden to check that two objects are equivalent.

In Kotlin, the == operator automatically invokes the `equals` function. The `open` class `Any` declares the `equals` function, as shown in Example 3-19, along with `hashCode` and `toString`.

Example 3-19. The declarations of `equals`, `hashCode`, and `toString` in `Any`

```
open class Any {
    open operator fun equals(other: Any?): Boolean

    open fun hashCode(): Int

    open fun toString(): String
}
```

The contract for `equals` requires that the implementation be reflexive, symmetric, and transitive, as well as consistent, and handle nulls appropriately. The contract for `hashCode` is that if two objects are equal by the `equals` function, they should have the same `hashCode` as well. The `hashCode` function should be overridden whenever the `equals` function is.

That said, how do you go about implementing a good `equals` function? One excellent example from the library is provided by the `KotlinVersion` class, whose `equals` function is shown in Example 3-20.

Example 3-20. The `equals` function in `KotlinVersion`

```
override fun equals(other: Any?): Boolean {
    if (this === other) return true
    val otherVersion = (other as? KotlinVersion) ?: return false
    return this.version == otherVersion.version
}
```

Note the simple elegance of this implementation, which takes advantage of several Kotlin features:

- First, it checks reference equality by using ===.
- Then it uses the safe casting operator, `as?`, which either casts the argument as the desired type or returns `null`.
- If the safe cast returns `null`, the Elvis operator (`?:`) then returns `false` because if the instances aren't of the same class, they can't be equal.
- Finally, the last line checks whether the `version` property of the current instance (not shown) is equivalent (using the == operator) to the same property in the other object, and returns the result.

In three lines, this covers all the required cases. For completeness, the implementation of `hashCode` is simply as follows:

```
    override fun hashCode(): Int = version
```

This is interesting, but not as directly relevant if you're trying to understand how to write your own `equals` function. Say, for example, that you have a simple class called `Customer` with a string property called `name`. A consistent implementation of both `equals` and `hashCode` is shown in Example 3-21.

Example 3-21. Implementing `equals` and `hashCode` in `Customer`

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

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        val otherCustomer = (other as? Customer) ?: return false
        return this.name == otherCustomer.name
    }

    override fun hashCode() = name.hashCode()
}
```

Incidentally, if you let IntelliJ IDEA generate an `equals` and `hashCode` implementation for you, the result (using the Ultimate Edition version 2019.2) is shown in Example 3-22.

Example 3-22. Generated `equals` and `hashCode` functions by IntelliJ IDEA

```
class Customer(val name: String) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false

        other as Customer

        if (name != other.name) return false

        return true
    }

    override fun hashCode(): Int {
        return name.hashCode()
    }
}
```

The difference is that the generated `equals` function checks that the `javaClass` properties on the `KClass` are equivalent before casting, using the `as` operator, and then relies on the resulting *smart cast* to check the `name` properties. This is essentially equivalent to the procedure described previously, only a bit more verbose.

Data classes have their own autogenerated implementations of `equals` and `hashCode` (as well as `toString`, `copy`, and component methods). Here, however, you can see how easy it is to implement your own versions.

See Also

Data classes are discussed in Recipe 3.3. The `KotlinVersion` class is shown in Recipe 11.1.

3.8 Creating a Singleton

Problem

You want to ensure that only one instance of a class is available.

Solution

Use the `object` keyword instead of `class`.

Discussion

The Singleton design pattern defines a mechanism for guaranteeing there is only one instance for a particular class. To define a singleton:

1. Declare all constructors of a class to be private.
2. Provide a `static` factory method that returns a reference to the class, instantiating it if necessary.

The Singleton pattern is controversial, because it is sometimes used in cases where a small number of instances could be used rather than one. Nevertheless, it is one of the fundamental design patterns defined in *Design Patterns* by Erich Gamma et al. (Addison-Wesley Professional), and it can be quite helpful in certain cases.

An example of a singleton in the Java standard library is given by the `Runtime` class. Say you want to know how many processors are available on a given platform. In Java, you can find out with the code in Example 3-23.

Example 3-23. Finding the number of processors

```kotlin
fun main() {
    val processors = Runtime.getRuntime().availableProcessors()
    println(processors)
}
```

The `getRuntime` method is the `static` method used to return the singleton instance of the class. Example 3-24 shows the relevant portion of the `java.lang.Runtime` class.

Example 3-24. `Runtime` implemented to use the Singleton pattern

```
public class Runtime {
    private static final Runtime currentRuntime = new Runtime();

    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}

// ...
}
```

The `Runtime` class contains a private, static, final instance of the class, which is eagerly instantiated where the `currentRuntime` attribute is declared. The only constructor is declared to be `private`, and the static factory method is `getRuntime`, as used in Example 3-24.

To implement a singleton in Kotlin, simply use the `object` keyword instead of `class`, as in Example 3-25. This is known as an *object declaration*.

Example 3-25. Defining a singleton in Kotlin

```
object MySingleton {
    val myProperty = 3

    fun myFunction() = "Hello"
}
```

If you decompile the generated bytecodes, you get a result similar to Example 3-26.

Example 3-26. Decompiled code for a singleton from `object`

```
public final class MySingleton {
   private static final int myProperty = 3;
   public static final MySingleton INSTANCE;  ❶

   private MySingleton() {                    ❷
   }

   public final int getMyProperty() {
      return myProperty;
```

```
    }

    public final void myFunction() {
        return "Hello";
    }


    static {
        MySingleton var0 = new MySingleton();  ❸
        INSTANCE = var0;
        myProperty = 3;
    }
}
```

❶ Generated `INSTANCE` property

❷ Private constructor

❸ Eager instantiation of singleton

When invoking code in the singleton, you can access the members from the object name, as you would for `static` members in Java. The member function and property become `static final` methods and attributes in the decompiled Java class, along with any required getter methods, and the properties are initialized in a `static` block, along with the class itself. Kotlin code to access the members is in Example 3-27.

Example 3-27. Accessing members of a singleton from Kotlin

```
MySingleton.myFunction()
MySingleton.myProperty
```

Accessing the singleton from Java uses the generated `INSTANCE` property and is shown in Example 3-28.

Example 3-28. Accessing members of a singleton from Java

```
MySingleton.INSTANCE.myFunction();
MySingleton.INSTANCE.getMyProperty();
```

A complication arises if you want your singleton to be instantiated with an argument. Say, for example, you're writing a database connection pool, which would be a natural singleton. The initial size of the pool would be a reasonable input parameter when generating the singleton. Unfortunately, a Kotlin `object` can't have a constructor, so there's no easy way to pass an argument to it.

The blog post "Kotlin Singletons with Argument" by Christophe Beyls discusses ways to handle arguments based on the implementation of the `lazy` delegate in the Kotlin library.

The referenced article gets into the complexities associated with making the singleton instantiation thread-safe, based on double-checked locking and `@Volatile`. See the article for details.

3.9 Much Ado About Nothing

Problem

You want to use the `Nothing` class idiomatically.

Solution

Use `Nothing` when a function never returns.

Discussion

You know Nothing, Jon Snow.

—Ygritte in Game of Thrones, *praising Jon Snow for reading this recipe*

There is a class in Kotlin called `Nothing`, whose entire implementation is given in Example 3-29.

Example 3-29. The `Nothing` implementation

```
package kotlin

public class Nothing private constructor()
```

The private constructor means the class cannot be instantiated outside the class, and as you can see, it isn't instantiated inside the class either. Therefore, there are no instances of `Nothing`. The documentation states that "you can use `Nothing` to represent a value that never exists."

The `Nothing` class arises naturally in two circumstances. The first occurs when a function body consists entirely of throwing an exception, as in Example 3-30.

Example 3-30. Throwing an exception in Kotlin

```
fun doNothing(): Nothing = throw Exception("Nothing at all")
```

The return type must be stated explicitly, and since the method never returns (it throws an exception instead), the return type is `Nothing`.

That is virtually guaranteed to be a source of confusion for existing Java developers. In Java, if a method throws an exception of any type, the return type on the method doesn't change. Exception handling is completely outside the normal flow of execution, but you don't have to change the return type on a method to account for it. The type system in Kotlin, however, has different requirements.

The other context in which `Nothing` arises occurs when you assign a variable to null and don't give it an explicit type, as in Example 3-31.

Example 3-31. A variable assigned to null without an explicit type

```
val x = null
```

The type of `x` is inferred to be `Nothing?`, because it's obviously nullable (it was assigned `null`, after all) and the compiler has no other information about it.

To really make matters interesting, consider this fact: in Kotlin, the `Nothing` class is actually a subtype of every other type.

To see why that is necessary, consider an `if` statement that can throw an exception, as in Example 3-32.

Example 3-32. An `if` statement that can throw an exception

```
val x = if (Random.nextBoolean()) "true" else throw Exception("nope")
```

The type of `x` is inferred to be either `String`, `Comparable<String>`, `CharSequence`, `Serializable`, or even `Any`, based on the string assigned when a true boolean is generated by the `Random.nextBoolean` function. The `else` clause returns a value of type `Nothing`, and since `Nothing` is a subtype of every type, performing a Boolean "and" with it and any other type is the other type.

Perhaps the following example will be clearer. The remainder of any number when divided by 3 must be either 0, 1, or 2. Therefore, the `when` statement in Example 3-33 shouldn't need an `else` clause, but the compiler doesn't know that.

Example 3-33. Remainder modulo 3

```
for (n in 1..10) {
    val x = when (n % 3) {
        0 -> "$n % 3 == 0"
        1 -> "$n % 3 == 1"
        2 -> "$n % 3 == 2"
        else -> throw Exception("Houston, we have a problem...")
    }
    assertTrue(x is string)
}
```

The `when` construct returns a value, so the compiler requires it to be exhaustive. The `else` condition should never happen, so it makes sense to throw an exception in that case. The return type on throwing an exception is `Nothing`, and since `String` is `String`, the compiler knows that the type of `x` is `String`.

> The `TODO` function (discussed in Recipe 11.9) returns `Nothing`, which makes sense because its implementation is to throw a `NotImplementedError`.

The `Nothing` class can be confusing, but once you know the use cases for it, it makes sense in context.

CHAPTER 4

Functional Programming

The term *functional programming* refers to a style of coding that favors immutability, is easy to make concurrent when using pure functions, uses transformations over looping, and uses filters over conditional statements. This book uses functional approaches throughout, especially in Chapters 5, 6, and 13. Many of the functions used by Kotlin in functional programming, like `map` and `filter`, are discussed where they arise in individual recipes of those chapters and others.

This chapter contains recipes that involve functional features that are either unique to Kotlin (as opposed to Java), like tail recursion, or are implemented somewhat differently, like the `fold` and `reduce` functions.

4.1 Using fold in Algorithms

Problem

You want to implement an iterative algorithm in a functional way.

Solution

Use the `fold` function to reduce a sequence or collection to a single value.

Discussion

The `fold` function is a reduction operation that can be applied to arrays or iterables. The syntax of the function is given by the following:

```
inline fun <R> Iterable<T>.fold(
    initial: R,
    operation: (acc: R, T) -> R
): R
```

The same function is defined on `Array`, as well as all the typed arrays, like `IntArray`, `DoubleArray`, and so on.

The idea is that `fold` takes two parameters: an initial value for the accumulator, and a function of two arguments that returns a new value for the accumulator. The classic example of a `fold` operation is a sum. See Example 4-1.

Example 4-1. Summing integers by using `fold`

```
fun sum(vararg nums: Int) =
    nums.fold(0) { acc, n -> acc + n }
```

In this case, the initial value is 0, and the supplied lambda takes two arguments, the first of which is an accumulator. The second iterates over each value in the parameter list. The test in Example 4-2 shows that the result is correct.

Example 4-2. Testing the `sum` operation

```
@Test
fun `sum using fold`() {
    val numbers = intArrayOf(3, 1, 4, 1, 5, 9)
    assertEquals(numbers.sum(), sum(*numbers))
}
```

The result from the provided `sum` function is compared to using the direct `sum` function defined on `IntArray`. Although this shows that the operation works, it doesn't give much insight into how. For that, add a `print` statement to see the values as they go by, as in Example 4-3.

Example 4-3. The `sum` function that prints each value

```
fun sumWithTrace(vararg nums: Int) =
    nums.fold(0) { acc, n ->
        println("acc = $acc, n = $n")
        acc + n
    }
```

Invoking a test similar to the preceding one results in this:

```
acc =  0, n = 3
acc =  3, n = 1
acc =  4, n = 4
acc =  8, n = 1
acc =  9, n = 5
acc = 14, n = 9
```

The `acc` variable is initialized to the first argument in `fold`, the `n` variable takes on each element of the collection, and the result of the lambda, `acc + n`, is the new value of `acc` on each iteration.

The lambda itself is a *binary operator*, because the data types of the accumulator, each element of the collection, and the return value are all the same.

Although the first argument to `fold` is called `initial` and initializes the accumulator, technically it should be the identity value for the lambda operation.

As a more interesting example, consider computing the factorial of an integer. The factorial operation is easily expressed as a recursive operation, which you'll see again in Example 4-10:

```
    fun recursiveFactorial(n: Long): BigInteger =
        when (n) {
            0L, 1L -> BigInteger.ONE
            else -> BigInteger.valueOf(n) * recursiveFactorial(n - 1)
        }
```

This operation can be rewritten as an iterative operation using `fold`, as shown in Example 4-4.

Example 4-4. Implementing the factorial by using `fold`

```
fun factorialFold(n: Long): BigInteger =
    when(n) {
        0L, 1L -> BigInteger.ONE
        else -> (2..n).fold(BigInteger.ONE) { acc, i ->
            acc * BigInteger.valueOf(i) }
    }
```

The `when` condition checks the input argument for 0 or 1, and returns `BigInteger.ONE` in those cases. The `else` condition uses a range from 2 to the input number and applies a `fold` operation that starts at `BigInteger.ONE`. The accumulator in the lambda is set to the product of the previous accumulator and each value as it goes by. Again, although `BigInteger.ONE` is the initial value of the accumulator, it's also the identity value of the multiplication (binary) operation.

To give one more fascinating example of `fold`, consider computing Fibonacci numbers, where each value is the sum of the previous two. Example 4-5 shows how to implement that algorithm by using `fold`.

Example 4-5. Computing Fibonacci numbers by using `fold`

```
fun fibonacciFold(n: Int) =
    (2 until n).fold(1 to 1) { (prev, curr), _ ->
        curr to (prev + curr) }.second
```

In this case, the initial value of the accumulator is a `Pair` whose `first` and `second` values are both 1. Then the lambda is able to create a new value for the accumulator without caring which particular index is being computed, which is why an underscore (`_`) is used as a placeholder for that value. The lambda creates a new `Pair` by assigning the current value to the previous one, and making the new value of `curr` equal to the sum of the existing previous and current values. This process is repeated from 2 up to the desired index. In the end, the output value is the `second` property of the final `Pair`.

Another interesting feature of this example is that the accumulator is of a different type than the elements in the range. The accumulator is a `Pair`, while the elements are `Int` values.

Using `fold` like this shows it is far more powerful than as demonstrated in the typical `sum` example.

See Also

The factorial problem is also addressed in Recipe 4.3. Using `reduce` instead of `fold` is part of Recipe 4.2.

4.2 Using the reduce Function for Reductions

Problem

You want to perform a reduction on a non-empty collection of values, but don't need to set an initial value for the accumulator.

Solution

Use the `reduce` operation rather than `fold`.

Discussion

The `reduce` function is similar to the `fold` function discussed in Recipe 4.1. Its signature on `Iterable` is as follows:

```
inline fun <S, T : S> Iterable<T>.reduce(
    operation: (acc: S, T) -> S
): S
```

The `reduce` function is almost exactly the same as the `fold` function, and it's used for the same purpose. Its biggest difference is that it does not have an argument that provides an initial value for the accumulator. The accumulator is therefore initialized with the first value from the collection.

Example 4-6 shows an implementation of `reduce` in the standard library.

Example 4-6. Implementation of the `reduce` function

```
public inline fun IntArray.reduce(
    operation: (acc: Int, Int) -> Int): Int {
    if (isEmpty())                                  ❶
        throw UnsupportedOperationException(
            "Empty array can't be reduced.")
    var accumulator = this[0]                       ❷
    for (index in 1..lastIndex) {
        accumulator = operation(accumulator, this[index])
    }
    return accumulator
}
```

❶ Empty collections result in an exception

❷ Accumulator initialized to first element of collection

The `reduce` function can therefore be used only when it is appropriate to initialize the accumulator with the first value of the collection. An example is an implementation of the `sum` operation, similar to that shown previously in Example 4-1, and shown here in Example 4-7.

Example 4-7. Implementing `sum` using `reduce`

```
fun sumReduce(vararg nums: Int) =
    nums.reduce { acc, i -> acc + i }
```

If this function is invoked with several arguments, the first argument initializes the accumulator, and the other values are added to it one by one. If this function is invoked with no arguments, it would throw an exception, as shown in Example 4-8.

Example 4-8. Testing the `sum` function implemented with `reduce`

```
@Test
fun `sum using reduce`() {
    val numbers = intArrayOf(3, 1, 4, 1, 5, 9)
    assertAll(
        { assertEquals(numbers.sum(), sumReduce(*numbers)) },   ❶
        { assertThrows<UnsupportedOperationException> {
            sumReduce()
        }         ❷
    })
}
```

❶ Validation for array of `Int`

❷ Throws exception for no arguments

There is another way that using `reduce` can go wrong. Say you want to modify all the input values before adding them together. For example, if you want to double each number before adding it to the sum, you might implement the function as in Example 4-9.

Example 4-9. Doubling values before adding

```
fun sumReduceDoubles(vararg nums: Int) =
    nums.reduce { acc, i -> acc + 2 * i }
```

Summing the values {3, 1, 4, 1, 5, 9}, while printing out the values of the accumulator and the `i` variable along the way, results in the following:

```
acc=3, i=1
acc=5, i=4
acc=13, i=1
acc=15, i=5
acc=25, i=9

org.opentest4j.AssertionFailedError:
Expected :46
Actual   :43
```

The result is off because the first value in the list, 3, was used to initialize the accumulator and therefore wasn't doubled. For this operation, it would be more appropriate to use `fold` rather than `reduce`.

Use `reduce` only when it is acceptable to initialize the accumulator with the first value of the collection and no additional processing is done on the other values.

In Java, streams have a method called `reduce` that has two overloads—one that takes just a binary operator (a lambda is used here), and one that includes an initial value as provided to `fold`. Also, when you call the overload that does not have an initial value, the return type is `Optional`, so rather than throwing an exception on an empty stream, Java returns an empty `Optional`.

The designers of the Kotlin library decided to implement those capabilities as separate functions, and the `reduce` operation throws an exception on an empty collection. If you come from a Java background, keep those differences in mind when deciding which function to use.

See Also

Recipe 4.1 discusses the `fold` function.

4.3 Applying Tail Recursion

Problem

You have a recursive process and want to minimize the memory required to execute it.

Solution

Express your algorithm by using tail recursion and add the `tailrec` keyword to your function.

Discussion

Developers tend to favor iterative algorithms when implementing a function, because they often are easier to understand and code. Some procedures, however, are most easily expressed recursively. As a trivial example, consider computing the factorial of a number, as in Example 4-10.

Example 4-10. Implementing a factorial as a recursive function

```kotlin
fun recursiveFactorial(n: Long): BigInteger =
    when (n) {
        0L, 1L -> BigInteger.ONE
        else -> BigInteger.valueOf(n) * recursiveFactorial(n - 1)
    }
```

The idea is pretty simple: the factorials of 0 and 1 are both equal to 1(`0! == 1, 1! == 1`) and for every number greater than 1, the factorial is equal to the product of that number times the factorial of one less than the number. Since the result is going to grow quickly, the code here uses the `BigInteger` class for the return type, even though the argument is a long value.

Each new recursive call adds a new frame to the call stack, so eventually the process exceeds available memory. A sample test case to demonstrate this is given in Example 4-11.

Example 4-11. Testing the recursive factorial implementation

```
@Test
fun `check recursive factorial`() {
    assertAll(
        { assertThat(recursiveFactorial(0), `is`(BigInteger.ONE)) },
        { assertThat(recursiveFactorial(1), `is`(BigInteger.ONE)) },
        { assertThat(recursiveFactorial(2), `is`(BigInteger.valueOf(2))) },
        { assertThat(recursiveFactorial(5), `is`(BigInteger.valueOf(120))) },
        { assertThrows<StackOverflowError> { recursiveFactorial(10_000) }} ❶
    )
}
```

❶ High-enough number results in a `StackOverflowError`

The JVM crashes with a `StackOverflowError` once the process hits the stack size limit (which defaults to 1,024 kilobytes on OpenJDK 1.8).

The approach known as *tail recursion* is a special case of recursion that can be implemented without adding a new stack frame to the call stack. To do this, rewrite the algorithm so that the recursive call is the last operation performed. That way, the current stack frame can be reused.

In Kotlin, a factorial version suitable for tail recursion is shown in Example 4-12.

Example 4-12. Implementing a factorial with a tail call algorithm

```
@JvmOverloads                                               ❶
tailrec fun factorial(n: Long,                              ❷
                      acc: BigInteger = BigInteger.ONE): BigInteger =
    when (n) {
        0L -> BigInteger.ONE
        1L -> acc
        else -> factorial(n - 1, acc * BigInteger.valueOf(n)) ❸
    }
```

❶ Annotation allows invoking the function from Java with only one argument

❷ Uses the `tailrec` keyword

❸ Tail-recursive call

In this case, the factorial function needs a second argument that acts as the accumulator for the factorial computation. That way, the last evaluated expression can call itself with a smaller number and an increased accumulator.

The second argument is assigned a default value of `BigInteger.ONE`, and since it is a default value, the `factorial` function can be called without it. Because Java doesn't have default function arguments, adding the `@JvmOverloads` annotation means the process works when invoked from Java too.

The key piece of the puzzle, however, is the addition of the `tailrec` keyword. Without that, the compiler would not know to optimize the recursion. With the keyword applied, the function becomes a fast and efficient loop-based version instead.

The `tailrec` keyword tells the compiler to optimize away the recursive call. The same algorithm expressed in Java will still be recursive, with the same memory limitations.

The tests in Example 4-13 show that the function can now be called for numbers so large that the test just verifies the number of digits in the answer.

Example 4-13. Testing the tail-recursion implementation

```
@Test
fun `factorial tests`() {
    assertAll(
        { assertThat(factorial(0), `is`(BigInteger.ONE)) },
        { assertThat(factorial(1), `is`(BigInteger.ONE)) },
        { assertThat(factorial(2), `is`(BigInteger.valueOf(2))) },
        { assertThat(factorial(5), `is`(BigInteger.valueOf(120))) },
        // ...
        { assertThat(factorial(15000).toString().length, `is`(56130)) },   ❶
        { assertThat(factorial(75000).toString().length, `is`(333061)) }   ❶
    )
}
```

❶ Checks number of digits in result

If you generate the bytecodes from the Kotlin implementation and decompile them back to Java, the result is similar to Example 4-14.

Example 4-14. Decompiled Java from the Kotlin bytecodes (rewritten)

```
public static final BigInteger factorial(long n, BigInteger acc) {
    while(true) {
        BigInteger result;
        if (n == 0L) {
            result = BigInteger.ONE;
         } else {
            if (n != 1L) {
               result = result.multiply(BigInteger.valueOf(n));
               n = n - 1L;
               continue;
            }
        }
        return result;
    }
}
```

The recursive call has been refactored by the compiler into an iterative algorithm using a `while` loop.

To summarize, the requirements for a function to be eligible for the `tailrec` modifier are as follows:

- The function must call itself as the last operation it performs.
- You cannot use `tailrec` inside try/catch/finally blocks.
- Tail recursion is supported only on the JVM backend.

CHAPTER 5
Collections

As in Java, Kotlin uses typed collections to hold multiple objects. Unlike Java, Kotlin adds many interesting methods directly to the collection classes, rather than going through a stream intermediary.

Recipes in this chapter discuss ways to process both arrays and collections, ranging from sorting and searching, to providing read-only views, to accessing windows of data, and more.

5.1 Working with Arrays

Problem

You want to create and populate arrays in Kotlin.

Solution

Use the `arrayOf` function to create them, and the properties and methods inside the `Array` class to work with the contained values.

Discussion

Virtually every programming language has arrays, and Kotlin is no exception. This book focuses on Kotlin running on the JVM, and in Java arrays are handled a bit differently than they are in Kotlin. In Java you instantiate an array using the keyword `new` and dimensioning the array, as in Example 5-1.

Example 5-1. Instantiating an array in Java

```
String[] strings = new String[4];
strings[0] = "an";
strings[1] = "array";
strings[2] = "of";
strings[3] = "strings";

// or, more easily,
strings = "an array of strings".split(" ");
```

Kotlin provides a simple factory method called `arrayOf` for creating arrays, and while it uses the same syntax for accessing elements, in Kotlin `Array` is a class. Example 5-2 shows how the factory method works.

Example 5-2. Using the `arrayOf` factory method

```
val strings = arrayOf("this", "is", "an", "array", "of", "strings")
```

You can also use the factory method `arrayOfNulls` to create (as you might guess) an array containing only nulls, as in Example 5-3.

Example 5-3. Creating an array of nulls

```
val nullStringArray = arrayOfNulls<String>(5)
```

It's interesting that even though the array contains only `null` values, you still have to choose a particular data type for it. After all, it may not contain nulls forever, and the compiler needs to know what type of reference you plan to add to it. The factory method `emptyArray` works the same way.

There is only one public constructor in the `Array` class. It takes two arguments:

- `size` of type `Int`
- `init`, a lambda of type `(Int) -> T`

The lambda is invoked on each index when creating the array. For example, to create an array of strings containing the first five integers squared, see Example 5-4.

Example 5-4. Array of strings containing the squares of 0 through 4

```
val squares = Array(5) { i -> (i * i).toString() } ❶
```

❶ Resulting array is `{"0", "1", "4", "9", "16"}`

The `Array` class declares public operator methods `get` and `set`, which are invoked when you access elements of the array using square brackets, as in `squares[1]`.

Kotlin has specialized classes to represent arrays of primitive types to avoid the cost of autoboxing and unboxing. The functions `booleanArrayOf`, `byteArrayOf`, `shortArrayOf`, `charArrayOf`, `intArrayOf`, `longArrayOf`, `floatArrayOf`, and `doubleArrayOf` create the associated types (`BooleanArray`, `ByteArray`, `ShortArray`, etc.) exactly the way you would expect.

Even though Kotlin doesn't have explicit primitives, the generated bytecodes use Java wrapper classes like `Integer` and `Double` when the values are nullable, and primitive types like `int` and `double` if not.

Many of the extension methods on arrays are the same as their counterparts on collections, which are discussed in the rest of this chapter. A couple are unique to arrays, however. For example, if you want to know the valid index values for a given array, use the property `indices`, as in Example 5-5.

Example 5-5. Getting the valid index values from an array

```
@Test
fun `valid indices`() {
    val strings = arrayOf("this", "is", "an", "array", "of", "strings")
    val indices = strings.indices
    assertThat(indices, contains(0, 1, 2, 3, 4, 5))
}
```

Normally you iterate over an array using the standard for-in loop, but if you want the index values as well, use the function `withIndex`.

```
    fun <T> Array<out T>.withIndex(): Iterable<IndexedValue<T>>

    data class IndexedValue<out T>(public val index: Int,
                                   public val value: T)
```

The class `IndexedValue` is the data class shown, with properties called `index` and `value`. Use it as shown in Example 5-6.

Example 5-6. Accessing array values using `withIndex`

```
@Test
fun `withIndex returns IndexValues`() {
    val strings = arrayOf("this", "is", "an", "array", "of", "strings")
    for ((index, value) in strings.withIndex()) {       ❶
        println("Index $index maps to $value")           ❷
        assertTrue(index in 0..5)
    }
}
```

❶ Call `withIndex`

❷ Access individual indices and values

The results printed to standard output are:

```
    Index 0 maps to this
    Index 1 maps to is
    Index 2 maps to an
    Index 3 maps to array
    Index 4 maps to of
    Index 5 maps to strings
```

In general, Kotlin arrays behave the same way arrays in other languages do.

5.2 Creating Collections

Problem

You want to generate a list, set, or map.

Solution

Use one of the functions designed to produce either an unmodifiable collection, like `listOf`, `setOf`, and `mapOf`, or their mutable equivalents, `mutableListOf`, `mutableSetOf`, and `mutableMapOf`.

Discussion

If you want an immutable view of a collection, the *kotlin.collections* package provides a series of utility functions for doing so.

One example is `listOf(vararg elements: T): List<T>`, whose implementation is shown in Example 5-7.

Example 5-7. Implementation of the `listOf` function

```
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.size > 0) elements.asList() else emptyList()
```

The referenced `asList` function is an extension function on `Array` that returns a `List` that wraps the specified array. The resulting list is called immutable, but should more properly be considered read-only: you cannot add to nor remove elements from it, but if the contained objects are mutable, the list will appear to change.

The implementation of `asList` delegates to Java's `Arrays.asList`, which returns a read-only list.

Similar functions in the same package include the following:

- `listOf`
- `setOf`
- `mapOf`

Example 5-8 shows how to create lists and sets.

Example 5-8. Creating "immutable" lists, sets, and maps

```
var numList = listOf(3, 1, 4, 1, 5, 9)                   ❶
var numSet = setOf(3, 1, 4, 1, 5, 9)                     ❷
// numSet.size == 5                                      ❸
var map = mapOf(1 to "one", 2 to "two", 3 to "three")    ❹
```

❶ Creates an unmodifiable list

❷ Creates an unmodifiable set

❸ Set does not contain duplicates

❹ Creates a map from `Pair` instances

By default, Kotlin collections are "immutable," in the sense that they do not support methods for adding or removing elements. If the elements themselves can be modified, the collection can appear to change, but only read-only operations are supported on the collection itself.

Methods to modify collections are in the "mutable" interfaces, provided by the factory methods:

- `mutableListOf`
- `mutableSetOf`
- `mutableMapOf`

Example 5-9 shows the analogous mutable examples.

Example 5-9. Creating mutable lists, sets, and maps

```
var numList = mutableListOf(3, 1, 4, 1, 5, 9)
var numSet = mutableSetOf(3, 1, 4, 1, 5, 9)
var map = mutableMapOf(1 to "one", 2 to "two", 3 to "three")
```

The implementation of the `mapOf` function in the standard library is shown here:

```
    public fun <K, V> mapOf(vararg pairs: Pair<K, V>): Map<K, V> =
        if (pairs.size > 0)
            pairs.toMap(LinkedHashMap(mapCapacity(pairs.size)))
        else emptyMap()
```

The argument to the `mapOf` function is a variable argument list of `Pair` instances, so the infix `to` operator function is used to create the map entries. A similar function is used to create mutable maps.

You can also instantiate classes that implement the `List`, `Set`, or `Map` interfaces directly, as shown in Example 5-10.

Example 5-10. Instantiating a linked list

```
@Test
internal fun `instantiating a linked list`() {
    val list = LinkedList<Int>()
    list.add(3)                           ❶
    list.add(1)
    list.addLast(999)                     ❶
    list[2] = 4                           ❷
    list.addAll(listOf(1, 5, 9, 2, 6, 5))
    assertThat(list, contains(3, 1, 4, 1, 5, 9, 2, 6, 5))
}
```

❶ The `add` method is an alias for `addLast`

❷ Array-type access invokes `get` or `set`

5.3 Creating Read-Only Views from Existing Collections

Problem

You have an existing mutable list, set, or map, and you want to create a read-only version of it.

Solution

To make a new, read-only collection, use the `toList`, `toSet`, or `toMap` methods. To make a read-only view on an existing collection, assign it to a variable of type `List`, `Set`, or `Map`.

Discussion

Consider a mutable list created with the `mutableList` factory method. The resulting list has methods like `add`, `remove`, and so on that allow the list to grow or shrink as desired:

```
val mutableNums = mutableListOf(3, 1, 4, 1, 5, 9)
```

There are two ways to create a read-only version of a mutable list. The first is to invoke the `toList` method, which returns a reference of type `List`:

```
@Test
fun `toList on mutableList makes a readOnly new list`() {
    val readOnlyNumList: List<Int> = mutableNums.toList() ❶
    assertEquals(mutableNums, readOnlyNumList)
    assertNotSame(mutableNums, readOnlyNumList)
}
```

❶ Explicit type shows result is a `List<T>`

The test shows that the return type from the `toList` method is `List<T>`, which represents an immutable list, so methods like `add` or `remove` are not available. The rest of the test shows that the method is creating a separate object, however, so while it has the same contents as the original, it doesn't represent the same objects anymore:

```
@Test
internal fun `modify mutable list does not change read-only list`() {
    val readOnly: List<Int> = mutableNums.toList()
    assertEquals(mutableNums, readOnly)

    mutableNums.add(2)
    assertThat(readOnly, not(contains(2)))
}
```

If you want a read-only view of the same contents, assign the mutable list to a reference of type `List`, as shown in Example 5-11.

Example 5-11. Creating a read-only view of the mutable list

```
@Test
internal fun `read-only view of a mutable list`() {
    val readOnlySameList: List<Int> = mutableNums ❶
    assertEquals(mutableNums, readOnlySameList)
    assertSame(mutableNums, readOnlySameList)

    mutableNums.add(2)
    assertEquals(mutableNums, readOnlySameList)
    assertSame(mutableNums, readOnlySameList)     ❷
}
```

❶ Assigns mutable to reference of type `List`

❷ Still the same underlying object

This time, the mutable list is assigned to a reference of type `List`. Not only is the result still the same object, but if the underlying mutable list is modified, the read-only view shows the updated values. You can't modify the list from the read-only reference, but it is attached to the same object as the original.

As you might expect, the `toSet` and `toMap` functions work the same way, as does assigning mutable sets and maps to references of type `Set` or `Map`.

5.4 Building a Map from a Collection

Problem

You have a list of keys and want to build a map by associating each key with a generated value.

Solution

Use the `associateWith` function by supplying a lambda to be executed for each key.

Discussion

Say you have a set of keys and want to map each of them to a generated value. One way to do that is to use the `associate` function, as in Example 5-12.

Example 5-12. Using `associate` to generate values

```
val keys = 'a'..'f'
val map = keys.associate { it to it.toString().repeat(5).capitalize() }
println(map)
```

Executing this snippet results in the following:

```
{a=Aaaaa, b=Bbbbb, c=Ccccc, d=Ddddd, e=Eeeee}
```

The `associate` function is an inline extension function on `Iterable<T>` that takes a lambda that transforms `T` into a `Pair<K,V>`. In this example, the `to` function is an infix function that produces a `Pair` from the left- and right-side arguments.

This works, but in Kotlin 1.3 a new function was added called `associateWith` that simplifies the code. Example 5-13 shows the previous code reworked with `associateWith`.

Example 5-13. Using `associateWith` to generate values

```kotlin
val keys = 'a'..'f'
val map = keys.associateWith { it.toString().repeat(5).capitalize() }
println(map)
```

The result is the same, but the argument now is a function that produces a `String` value rather than a `Pair<Char, String>`.

Both examples produce the same result, but the `associateWith` function is slightly simpler to write and understand.

5.5 Returning a Default When a Collection Is Empty

Problem

As you are processing a collection, you filter out all the elements but want to return a default response.

Solution

Use the `ifEmpty` and `ifBlank` functions to return a default when a collection is empty or a string is blank.

Discussion

Say you have a data class called `Product` that wraps a name, a price, and a boolean field to indicate whether the product is on sale, as in Example 5-14.

Example 5-14. Data class for a product

```kotlin
data class Product(val name: String,
                   var price: Double,
                   var onSale: Boolean = false)
```

If you have a list of products and you want the names of the products that are on sale, you could do a simple filtering operation, as follows:

```
    fun namesOfProductsOnSale(products: List<Product>) =
        products.filter { it.onSale }
            .map { it.name }
            .joinToString(separator = ", ")
```

The idea is to take a list of products, filter them by the boolean `onSale` property, and map them to just the names, which then are joined into a single string. The problem is that if no products are on sale, the filter will return an empty collection, which will then be converted into an empty string.

If you would rather return a specific string when the result is empty, you can use a function called `ifEmpty` on both `Collection` and `String`. Example 5-15 shows how to use either one.

Example 5-15. Using `ifEmpty` on `Collection` and `String`

```
fun onSaleProducts_ifEmptyCollection(products: List<Product>) =
    products.filter { it.onSale }
        .map { it.name }
        .ifEmpty { listOf("none") }          ❶
        .joinToString(separator = ", ")

fun onSaleProducts_ifEmptyString(products: List<Product>) =
        products.filter { it.onSale }
            .map { it.name }
            .joinToString(separator = ", ")
            .ifEmpty { "none" }              ❷
```

❶ Supplies default on empty collection

❷ Supplies default on empty string

In either case, a collection of products that are not on sale will return the string `"none"` as shown in the tests in Example 5-16.

Example 5-16. Testing products

```
class IfEmptyOrBlankKtTest {
    private val overthruster = Product("Oscillation Overthruster", 1_000_000.0)
    private val fluxcapacitor = Product("Flux Capacitor", 299_999.95, onSale = true)
    private val tpsReportCoverSheet = Product("TPS Report Cover Sheet", 0.25)

    @Test
    fun productsOnSale() {
        val products = listOf(overthruster, fluxcapacitor, tpsReportCoverSheet)
        assertAll( "On sale products",
            { assertEquals("Flux Capacitor",
                onSaleProducts_ifEmptyCollection(products)) },
            { assertEquals("Flux Capacitor",
                onSaleProducts_ifEmptyString(products)) })
    }

    @Test
    fun productsNotOnSale() {
        val products = listOf(overthruster, tpsReportCoverSheet)

        assertAll( "No products on sale",
            { assertEquals("none", onSaleProducts_ifEmptyCollection(products)) },
            { assertEquals("none", onSaleProducts_ifEmptyString(products)) })
    }
}
```

Java in version 8 added a class called `Optional<T>`, which is often used as a return type wrapper when a query may legitimately return a null or empty value. Kotlin supports this as well, but it's easy enough to return a specific value instead by using the `ifEmpty` function.

5.6 Restricting a Value to a Given Range

Problem

Given a value, you want to return it if it is contained in a specified range, or return the minimum or maximum of the range if not.

Solution

Use the `coerceIn` function on ranges, either with a range argument or specified min and max values.

Discussion

There are two overloads of the `coerceIn` function for ranges: one that takes a closed range as an argument, and one that takes min and max values.

For the first variation, consider an integer range from 3 to 8, inclusive. The test in Example 5-17 shows that `coerceIn` returns the value if it is contained in the range, or the boundaries if not.

Example 5-17. Coercing a value into a range

```
@Test
fun `coerceIn given a range`() {
    val range = 3..8

    assertThat(5, `is`(5.coerceIn(range)))
    assertThat(range.start, `is`(1.coerceIn(range)))          ❶
    assertThat(range.endInclusive, `is`(9.coerceIn(range))) ❷
}
```

❶ `range.start` is 3

❷ `range.endInclusive` is 8

Likewise, if you have the min and max values you want, you don't have to create a range to use the `coerceIn` function, as Example 5-18 shows.

Example 5-18. Coercing a value with a min and max

```
@Test
fun `coerceIn given min and max`() {
    val min = 2
    val max = 6

    assertThat(5, `is`(5.coerceIn(min, max)))
    assertThat(min, `is`(1.coerceIn(min, max)))
    assertThat(max, `is`(9.coerceIn(min, max)))
}
```

This version returns the value if it is between min and max, and the boundary values if not.

5.7 Processing a Window on a Collection

Problem

Given a collection of values, you want to process them by using a small window that traverses the collection.

Solution

Use the `chunked` function if you want to divide the collection into equal parts, or the `windowed` function if you want a block that slides along the collection by a given interval.

Discussion

Given an iterable collection, the `chunked` function splits it into a list of lists, where each has the given size or smaller. The function can return the list of lists, or you can also supply a transformation to apply to the resulting lists. The signatures of the `chunked` function are as follows:

```
fun <T> Iterable<T>.chunked(size: Int): List<List<T>>

fun <T, R> Iterable<T>.chunked(
    size: Int,
    transform: (List<T>) -> R
): List<R>
```

This all sounds more complicated than it is in practice. For example, consider a simple range of integers from 0 to 10. The test in Example 5-19 breaks it into groups of three consecutive numbers, or computes their sums or averages.

Example 5-19. Breaking a list into sections and processing them

```
@Test
internal fun chunked() {
    val range = 0..10

    val chunked = range.chunked(3)
    assertThat(chunked, contains(listOf(0, 1, 2), listOf(3, 4, 5),
        listOf(6, 7, 8), listOf(9, 10)))

    assertThat(range.chunked(3) { it.sum() }, `is`(listOf(3, 12, 21, 19)))
    assertThat(range.chunked(3) { it.average() }, `is`(listOf(1.0, 4.0, 7.0, 9.5)))
}
```

The first call simply returns the `List<List<Int>>` consisting of `[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]`. The second and third calls provide a lambda to compute the sums of each list or the average of each list, respectively.

The `chunked` function is actually a special case of the `windowed` function. Example 5-20 shows how the implementation of `chunked` delegates to `windowed`.

Example 5-20. Implementation of `chunked` in the standard library

```
public fun <T> Iterable<T>.chunked(size: Int): List<List<T>> {
    return windowed(size, size, partialWindows = true)
}
```

The `windowed` function takes three arguments, two of which are optional:

`size`
: The number of elements in each window

`step`
: The number of elements to move forward on each step (defaults to 1)

`partialWindows`
: A boolean that defaults to `false` and tells whether to keep the last section if it doesn't have the required number of elements

The `chunked` function calls `windowed` with both the `size` and `step` parameters equal to the `chunked` argument, so it moves the window forward by exactly that size each time. You can use `windowed` directly, however, to change that.

An example is to compute a *moving average*. Example 5-21 shows how to use `windowed` both to behave the same way as `chunked` and to move the window forward by only one element each time.

Example 5-21. Computing a moving average in each window

```kotlin
@Test
fun windowed() {
    val range = 0..10

    assertThat(range.windowed(3, 3),
        contains(listOf(0, 1, 2), listOf(3, 4, 5), listOf(6, 7, 8)))

    assertThat(range.windowed(3, 3) { it.average() },
        contains(1.0, 4.0, 7.0))

    assertThat(range.windowed(3, 1),
        contains(
            listOf(0, 1, 2), listOf(1, 2, 3), listOf(2, 3, 4),
            listOf(3, 4, 5), listOf(4, 5, 6), listOf(5, 6, 7),
            listOf(6, 7, 8), listOf(7, 8, 9), listOf(8, 9, 10)))

    assertThat(range.windowed(3, 1) { it.average() },
        contains(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0))
```

The `chunked` and `windowed` functions are useful for processing time-series data in stages.

5.8 Destructuring Lists

Problem

You want to use destructuring to access elements of a list.

Solution

Assign the list to a group of at most five elements.

Discussion

Destructuring is the process of extracting values from an object by assigning them to a collection of variables.

Example 5-22 shows how you can assign the first few elements of a list to defined variables in one step.

Example 5-22. Destructuring elements from a list

```
val list = listOf("a", "b", "c", "d", "e", "f", "g")
val (a, b, c, d, e) = list
println("$a $b $c $d $e")
```

This code prints the string `a b c d e`, because the first five elements of the created list are assigned to the variables of the same name. This works because the `List` class has extension functions defined in the standard library called `componentN`, where *N* goes from 1 to 5, as shown in Example 5-23.

Example 5-23. The `component1` extension function on `List` (from the standard library)

```
/**
 * Returns 1st *element* from the collection.
 */
@kotlin.internal.InlineOnly
public inline operator fun <T> List<T>.component1(): T {
    return get(0)
}
```

Destructuring relies on the existence of `componentN` functions. The `List` class contains implementations for `component1`, `component2`, `component3`, `component4`, and `component5`, so the preceding code works.

Data classes automatically add the associated component methods for all of their defined attributes. If you define your own class (and don't make it a data class), you can manually define any needed component methods as well.

Destructuring is a convenient way to extract multiple elements from an object. At the moment, the `List` class defines component functions for the first five elements. That may change in later Kotlin versions.

5.9 Sorting by Multiple Properties

Problem

You want to sort a class by one property, and then equal values by a second property, and so on.

Solution

Use the `sortedWith` and `compareBy` functions.

Discussion

Say we have a simple data class called `Golfer`, with a sample collection shown in Example 5-24.

Example 5-24. A data class and some sample data

```
data class Golfer(val score: Int, val first: String, val last: String)

val golfers = listOf(
    Golfer(70, "Jack", "Nicklaus"),
    Golfer(68, "Tom", "Watson"),
    Golfer(68, "Bubba", "Watson"),
    Golfer(70, "Tiger", "Woods"),
    Golfer(68, "Ty", "Webb")
)
```

If you would like to sort the golfers by score, then sort equal scores by last name, and finally sort those equal scores and last names by first name, you can use the code in Example 5-25.

Example 5-25. Sorting the golfers by successive properties

```
val sorted = golfers.sortedWith(
    compareBy({ it.score }, { it.last }, { it.first })
)

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

The result is as follows:

```
    Golfer(score=68, first=Bubba, last=Watson)
    Golfer(score=68, first=Tom, last=Watson)
    Golfer(score=68, first=Ty, last=Webb)
    Golfer(score=70, first=Jack, last=Nicklaus)
    Golfer(score=70, first=Tiger, last=Woods)
```

The three golfers who shot a 68 appear before the two who scored 70. Within the 68s, both Watsons appear ahead of Webb, and in the 70s, Nicklaus is before Woods. For the golfers named Watson who both scored 68s, Bubba appears before Tom.

The full signatures of the `sortedWith` and `compareBy` functions are given by Example 5-26.

Example 5-26. The signature of `sortedWith` in the standard library

```
fun <T> Iterable<T>.sortedWith(
    comparator: Comparator<in T>
): List<T>

fun <T> compareBy(
    vararg selectors: (T) -> Comparable<*>?
): Comparator<T>
```

So the `sortedWith` function takes a `Comparator`, and the `compareBy` function produces a `Comparator`. What's interesting about `compareBy` is that you can provide a list of selectors, each of which extracts a `Comparable` property (note that the property's class has to implement the `Comparable` interface), and the function will create a `Comparator` that sorts by them in turn.

The `sortBy` and `sortWith` functions sort their elements in place, and therefore require mutable collections.

Another way to solve the same problem is to build the `Comparator` by using the `thenBy` function, which applies a comparison after the previous one. The same collection is sorted in this manner in Example 5-27.

Example 5-27. Chaining comparators together

```
val comparator = compareBy<Golfer>(Golfer::score)
    .thenBy(Golfer::last)
    .thenBy(Golfer::first)

golfers.sortedWith(comparator)
    .forEach(::println)
```

The result is the same as in the previous example.

5.10 Defining Your Own Iterator

Problem

You have a class that wraps a collection and you would like to iterate over it easily.

Solution

Define an operator function that returns an iterator, which implements both a `next` and a `hasNext` function.

Discussion

The Iterator design pattern has an implementation in Java by defining the `Iterator` interface. Example 5-28 provides the corresponding definition in Kotlin.

Example 5-28. Iterator interface in `kotlin.collections`

```
interface Iterator<out T> {
    operator fun next(): T
    operator fun hasNext(): Boolean
}
```

In Java, the for-each loop lets you iterate over any class that implements `Iterable`. In Kotlin, a similar constraint works on the for-in loop. Consider a data class called `Player` and a class called `Team`, as given in Example 5-29.

Example 5-29. `Player` and `Team` classes

```
data class Player(val name: String)
class Team(val name: String,
           val players: MutableList<Player> = mutableListOf()) {

    fun addPlayers(vararg people: Player) =
        players.addAll(people)

    // ... other functions as needed ...
}
```

A `Team` contains a mutable list of `Player` instances. If you have a team with several players and you want to iterate over the players, you need to access the `players` property, as in Example 5-30.

Example 5-30. Iterating over a team of players

```
val team = Team("Warriors")
team.addPlayers(Player("Curry"), Player("Thompson"),
    Player("Durant"), Player("Green"), Player("Cousins"))

for (player in team.players) { ❶
    println(player)
}
```

❶ Accesses `players` property in loop

This can be (slightly) simplified by defining an `operator` function called `iterator` on the `Team`. Example 5-31 shows how to do this as an extension function, and the resulting simplified loop.

Example 5-31. Iterating over the team directly

```
operator fun Team.iterator() : Iterator<Player> = players.iterator()

for (player in team) { ❶
    println(player)
}
```

❶ Can iterate over team

Either way, the output is as follows:

```
    Player(name=Curry)
    Player(name=Thompson)
    Player(name=Durant)
    Player(name=Green)
    Player(name=Cousins)
```

In reality, the idea is to make the `Team` class implement the `Iterable` interface, which includes the `abstract operator` function `iterator`. That means the alternative to writing the extension function is to modify `Team`, as shown in Example 5-32.

Example 5-32. Implementing the `Iterable` interface

```
class Team(val name: String,
           val players: MutableList<Player> = mutableListOf()) : Iterable<Player> {

    override fun iterator(): Iterator<Player> =
        players.iterator()

    // ... other functions as needed ...
}
```

The result is the same, except that now all the extension functions on `Iterable` are available on `Team`, so you can write code like that in Example 5-33.

Example 5-33. Using `Iterator` extension functions on `Team`

```
assertEquals("Cousins, Curry, Durant, Green, Thompson",
    team.map { it.name }.joinToString())
```

The `map` function here iterates over the players, so `it.name` represents each player's name. Other extension functions can be used in the same way.

5.11 Filtering a Collection by Type

Problem

You want to create a new collection of elements of a specified type from an existing group of mixed types.

Solution

Use the extension functions `filterIsInstance` or `filterIsInstanceTo`.

Discussion

Collections in Kotlin include an extension function called `filter` that takes a predicate, which can be used to extract elements satisfying any boolean condition, as in Example 5-34.

Example 5-34. Filtering a collection by type, with erasure

```
val list = listOf("a", LocalDate.now(), 3, 1, 4, "b")
val strings = list.filter { it is String }

for (s in strings) {
    // s.length  // does not compile; type is erased
}
```

Although the filtering operation works, the inferred type of the `strings` variable is `List<Any>`, so Kotlin does not smart cast the individual elements to type `String`.

You could add an `is` check or simply use the `filterIsInstance` function instead, as in Example 5-35.

Example 5-35. Using reified types

```
val list = listOf("a", LocalDate.now(), 3, 1, 4, "b")

val all = list.filterIsInstance<Any>()
val strings = list.filterIsInstance<String>()
val ints = list.filterIsInstance<Int>()
val dates = list.filterIsInstance(LocalDate::class.java)

assertThat(all, `is`(list))
assertThat(strings, containsInAnyOrder("a", "b"))
assertThat(ints, containsInAnyOrder(1, 3, 4))
assertThat(dates, contains(LocalDate.now()))
```

In this case, the `filterIsInstance` function uses reified types, so the resulting collections are of a known type, and you don't have to check the type before using its properties. The implementation of the `filterIsInstance` function in the library is shown here:

```
    public inline fun <reified R> Iterable<*>.filterIsInstance(): List<R> {
        return filterIsInstanceTo(ArrayList<R>())
    }
```

The `reified` keyword applied to an `inline` function preserves the type, so the returned type is `List<R>`.

The implementation calls the function `filterIsInstanceTo`, which takes a collection argument of a particular type and populates it with elements of that type from the original. That function can also be used directly, as in Example 5-36.

Example 5-36. Using reified types to populate a provided list

```
val list = listOf("a", LocalDate.now(), 3, 1, 4, "b")

val all = list.filterIsInstanceTo(mutableListOf())
val strings = list.filterIsInstanceTo(mutableListOf<String>())
val ints = list.filterIsInstanceTo(mutableListOf<Int>())
val dates = list.filterIsInstanceTo(mutableListOf<LocalDate>())

assertThat(all, `is`(list))
assertThat(strings, containsInAnyOrder("a", "b"))
assertThat(ints, containsInAnyOrder(1, 3, 4))
assertThat(dates, contains(LocalDate.now()))
```

The argument to the `filterIsInstanceTo` function is a `MutableCollection<in R>`, so by specifying the type of the desired collection, you populate it with the instances of that type.

5.12 Making a Range into a Progression

Problem

You want to iterate over a range, but the range does not contain simple integers, characters, or longs.

Solution

Create a progression of your own.

Discussion

In Kotlin, a range is created when you use the *double dot* operator, as in `1..5`, which instantiates an `IntRange`. A *range* is a closed interval, defined by two endpoints that are both included in the range.

The standard library adds an extension function called `rangeTo` to any generic type `T` that implements the `Comparable` interface. Example 5-37 provides its implementation.

Example 5-37. Implementation of the `rangeTo` function for `Comparable` types

```
operator fun <T : Comparable<T>> T.rangeTo(that: T): ClosedRange<T> =
    ComparableRange(this, that)
```

The class `ComparableRange` simply extends `Comparable`, defines `start` and `endInclusive` properties of type `T`, and overrides `equals`, `hashCode`, and `toString` functions appropriately. The return type on `rangeTo` is `ClosedRange`, which is a simple interface defined in Example 5-38.

Example 5-38. The `ClosedRange` interface

```
interface ClosedRange<T: Comparable<T>> {
    val start: T
    val endInclusive: T
    operator fun contains(value: T): Boolean =
        value >= start && value <= endInclusive
    fun isEmpty(): Boolean = start > endInclusive
}
```

The operator function `contains` lets you use the `in` infix function to check whether a value is contained inside the range.

All this means that you can create a range based on any class that implements `Comparable`, and the infrastructure to support it is already there. As an example, for `java.time.LocalDate`, see Example 5-39.

Example 5-39. Using `LocalDate` in a range

```
@Test
fun `LocalDate in a range`() {
    val startDate = LocalDate.now()
    val midDate = startDate.plusDays(3)
    val endDate = startDate.plusDays(5)

    val dateRange = startDate..endDate

    assertAll(
        { assertTrue(startDate in dateRange) },
        { assertTrue(midDate in dateRange) },
        { assertTrue(endDate in dateRange) },
        { assertTrue(startDate.minusDays(1) !in dateRange) },
        { assertTrue(endDate.plusDays(1) !in dateRange) }
    )
}
```

That's all well and good, but the surprising part comes when you try to iterate over the range:

```
    for (date in dateRange) println(it)         // compiler error!
    (startDate..endDate).forEach { /* ... */ }  // compiler error!
```

The problem is that a range is not a progression. A *progression* is simply an ordered sequence of values. Custom progressions implement the `Iterable` interface, just as the existing progressions `IntProgression`, `LongProgression`, and `CharProgression` in the standard library do.

To demonstrate how to create a progression, consider the classes in Example 5-40 and Example 5-41.

The code in this example is based on the 2017 DZone article by Grzegorz Ziemoński entitled, "What Are Kotlin Progressions and Why Should You Care?"

First, here is the `LocalDateProgression` class, which implements both `Iterable<LocalDate>` and `ClosedRange<LocalDate>` interfaces.

Example 5-40. A progression for `LocalDate`

```
import java.time.LocalDate

class LocalDateProgression(
    override val start: LocalDate,
    override val endInclusive: LocalDate,
    val step: Long = 1
) : Iterable<LocalDate>, ClosedRange<LocalDate> {

    override fun iterator(): Iterator<LocalDate> =
        LocalDateProgressionIterator(start, endInclusive, step)

    infix fun step(days: Long) = LocalDateProgression(start, endInclusive, days)
}
```

From the `Iterator` interface, the only function that must be implemented is `itera tor`. Here it instantiates the class `LocalDateProgressionIterator`, shown next. The `infix step` function instantiates the class with the proper increment in days. The `ClosedRange` interface, as shown in the preceding code, defines the `start` and `endIn clusive` properties, so they are overridden here in the primary constructor.

Example 5-41. The iterator for the `LocalDateProgression` class

```
import java.time.LocalDate

internal class LocalDateProgressionIterator(
    start: LocalDate,
    val endInclusive: LocalDate,
    val step: Long
) : Iterator<LocalDate> {

    private var current = start

    override fun hasNext() = current <= endInclusive

    override fun next(): LocalDate {
        val next = current
        current = current.plusDays(step)
        return next
    }
}
```

The `Iterator` interface requires overriding `next` and `hasNext`, as shown.

Finally, use an extension function to redefine the `rangeTo` function to return an instance of the progression:

```
    operator fun LocalDate.rangeTo(other: LocalDate) =
        LocalDateProgression(this, other)
```

Now `LocalDate` can be used to create a range that can be iterated over, as shown in the tests in Example 5-42.

Example 5-42. Tests for the `LocalDate` progression

```
@Test
fun `use LocalDate as a progression`() {
    val startDate = LocalDate.now()
    val endDate = startDate.plusDays(5)

    val dateRange = startDate..endDate
    dateRange.forEachIndexed { index, localDate ->
        assertEquals(localDate, startDate.plusDays(index.toLong()))
    }

    val dateList = dateRange.map { it.toString() }
    assertEquals(6, dateList.size)
}

@Test
fun `use LocalDate as a progression with a step`() {
    val startDate = LocalDate.now()
    val endDate = startDate.plusDays(5)

    val dateRange = startDate..endDate step 2
    dateRange.forEachIndexed { index, localDate ->
        assertEquals(localDate, startDate.plusDays(index.toLong() * 2))
    }

    val dateList = dateRange.map { it.toString() }
    assertEquals(3, dateList.size)
}
```

Using the double dot operator creates a range, which in this case supports iteration, which is used by the `forEachIndexed` function. In this example, creating a progression requires two classes and an extension function, but the pattern is easy enough to replicate for your own classes.

CHAPTER 6
Sequences

This chapter looks at Kotlin sequences, which are just like the streams introduced in Java version 1.8. Admittedly that's a useful statement only if you already know how Java streams work,[1] but the recipes in this chapter will highlight both their similarities and their differences.

With collections, processing is *eager*: when you invoke `map` or `filter` on a collection, every element in the collection is processed. Sequences, however, are *lazy*: when you use a sequence to process data, each element completes the entire pipeline before the next element is processed. This helps when you have a lot of data or if a short-circuiting operation, like `first`, lets you exit the sequence when a desired value is found.

6.1 Using Lazy Sequences

Problem

You want to process the minimum amount of data necessary to satisfy a certain condition.

Solution

Use a Kotlin sequence with a short-circuiting function.

1 The most famous example of an explanation like that is, "Monads are just monoids in the category of endofunctors," which may be true but is almost, but not quite, a completely useless statement.

Discussion

Kotlin adds extension functions to basic collections, so that `List` has functions like `map` and `filter`. Those functions are eager, however, meaning that they process every element in the collection.

Consider the following admittedly highly contrived problem: you want to take the numbers from 100 to 200 and double each one, and then find the first double that's evenly divisible by 3. One way to solve that problem is to use the function in Example 6-1.

Example 6-1. Finding the first double divisible by 3 (version 1)

```
(100 until 200).map { it * 2 } ❶
    .filter { it % 3 == 0 }     ❷
    .first()
```

❶ 100 computations

❷ Another 100 (bad design—can be improved)

The problem with this approach is that it's horribly inefficient. First, you double all 100 numbers in the range, then you perform the modulus operation on all 100 results, and then grab just the first element. Fortunately, there is an overload of the `first` function that takes a predicate (a lambda of one argument that returns a boolean), as shown in Example 6-2.

Example 6-2. Finding the first double divisible by 3 (version 2)

```
(100 until 200).map { it * 2 } ❶
    .first { it % 3 == 0 }      ❷
```

❶ 100 computations

❷ Only 3 computations

This version of the `first` method uses a loop to process each element of the collection, but stops when it hits the first one that satisfies the predicate. That process is known as *short-circuiting*—processing only as much data as needed until a condition is reached. If you forget to use the overloaded version of `first`, however, you're back to doing much more work than needed.

Kotlin sequences process data differently. Example 6-3 is much better still.

Example 6-3. Finding the first double divisible by 3 (best)

```
(100 until 2_000_000).asSequence()  ❶
    .map { println("doubling $it"); it * 2 }
    .filter { println("filtering $it"); it % 3 == 0 }
    .first()
```

❶ Converts the range into a sequence

The function this time executes only six operations before returning the right answer:

```
doubling 100
filtering 200
doubling 101
filtering 202
doubling 102
filtering 204
```

For sequences, it doesn't matter whether you use the `filter` function shown or the other overload of `first`. The latter may be simpler, but either way, only six calculations are done, because each element of the sequence is processed by the entire pipeline completely before moving to the next one. Note that just to prove a point, the upper limit of the sequence this time was changed to two million, which didn't affect the resulting behavior at all.

Incidentally, the `first` function used here throws an exception if the sequence is empty. If that might happen, consider using `firstOrNull` instead.

The API for `Sequence` contains the same functions as that for `Collection`, but the operations fall into two categories: intermediate and terminal. *Intermediate operations*, like `map` and `filter`, return a new sequence. *Terminal operations*, like `first` or `toList`, return anything else. The key is that without a terminal operation, a sequence will not process any data.

A sequence processes data only if the pipeline of chained functions called on it ends with a terminal operation.

Unlike Java streams, Kotlin sequences can be iterated multiple times, though some cannot and are documented as such.

6.2 Generating Sequences

Problem

You want to generate a sequence of values.

Solution

Use `sequenceOf` if you already have the elements, use `asSequence` if you already have an `Iterable`, or use a sequence generator otherwise.

Discussion

The first two options are trivial. The `sequenceOf` function works the same way `arrayOf`, `listOf`, or any of the other related functions work. The `asSequence` function converts an existing `Iterable` (most likely a list or other collection) into a `Sequence`. Both of those options are shown in Example 6-4.

Example 6-4. Creating sequences when you already have the values

```
val numSequence1 = sequenceOf(3, 1, 4, 1, 5, 9)
val numSequence2 = listOf(3, 1, 4, 1, 5, 9).asSequence()
```

Both of these statements produce a `Sequence<Int>`, either from the given values or from the provided list.

Life gets interesting when you have to generate the elements in the sequence, which may even be infinite. To see that in action, first consider Example 6-5, which is an extension function on `Int` that determines whether a number is prime by dividing by each number from 2 up to the square root of the number, looking for any value that divides it evenly.

Example 6-5. Checking whether an `Int` is prime

```
import kotlin.math.ceil
import kotlin.math.sqrt

fun Int.isPrime() =
    this == 2 || (2..ceil(sqrt(this.toDouble())).toInt())
        .none { divisor -> this % divisor == 0 }
```

This function first checks to see whether the number is 2. If not, it creates a range from 2 up to the square root of the given number, rounded up to the next integer. For each number in that range, the `none` function returns `true` only if none of the values evenly divide the original number.

Test cases to verify the behavior of the `isPrime` function are included in the source code repository for this book.

Now say you have a particular integer and you want to know the next prime number that comes after it. As an example, given 6, the next prime is 7. Given 182, the next prime after that is 191. For 9,973, the next prime number is 10,007. What makes that problem interesting is that there's no way to know how many numbers you're going to need to check before you find the next prime. That makes a natural application for a sequence. An implementation of the `nextPrime` function is shown in Example 6-6.

Example 6-6. Finding the next prime after a given integer

```
fun nextPrime(num: Int) =
    generateSequence(num + 1) { it + 1 }   ❶
        .first(Int::isPrime)               ❷
```

❶ Starts one past the given number and iterates by 1

❷ Returns the first prime value

The `generateSequence` function takes two arguments: an initial value, and a function to produce the next value in the sequence. Its signature is given by the following:

```
fun <T : Any> generateSequence(
    seed: T?,
    nextFunction: (T) -> T?
): Sequence<T>
```

The seed in this case is the integer right after the provided one, and the function simply increments by one. By the normal Kotlin idiom, the lambda is placed after the parentheses in the `generateSequence` function. The `first` function on a sequence returns the first value that satisfies the supplied lambda, which in this case is a reference to the `isPrime` extension function.

In this case, the `nextPrime` function generates an infinite sequence of integers, evaluating them one by one until it finds the first prime. The `first` function returns a value rather than a sequence, so it is a *terminal* operation. Without a terminal operation, no values are processed by the sequence. In this case, the `first` operation is given a lambda known as a `predicate` (because it returns a boolean value), and the sequence keeps producing values until the predicate is satisfied.

See Also

Infinite sequences are also examined in Recipe 6.3.

6.3 Managing Infinite Sequences

Problem

You need a portion of an infinite sequence.

Solution

Use a sequence generator that returns a null, or use one of the sequence functions such as `takeWhile`.

Discussion

Sequences, like Java streams, have intermediate operations and terminal operations. An intermediate operation returns a new sequence, and a terminal operation returns anything else. When you create a pipeline of function calls on a sequence, no data is pulled through the sequence until you have a terminal operation.

The function `firstNPrimes`, shown in Example 6-7, calculates the first `N` prime numbers, starting from 2. The sample shown here leverages the `nextPrime` function given in Example 6-6 and described in Recipe 6.2, and repeated here for convenience:

```
fun nextPrime(num: Int) =
    generateSequence(num + 1) { it + 1 }
        .first(Int::isPrime)
```

This sequence uses the `isPrime` extension function from the same recipe.

Once again, there's no way to know ahead of time how many numbers need to be examined in order to find that many primes, so a sequence is a natural way to solve the problem.

Example 6-7. Finding the first `N` prime numbers

```
fun firstNPrimes(count: Int) =
    generateSequence(2, ::nextPrime)  ❶
        .take(count)                  ❷
        .toList()                     ❸
```

❶ Infinite sequence of primes, starting from 2

❷ Intermediate operation to retrieve only the count requested

❸ Terminal operation

The sequence generated is infinite. The `take` function is a stateless, intermediate operation that returns a sequence consisting of only the first `count` values. If you simply execute this function without the `toList` function at the end, no primes are computed. All you have is a sequence, but no values. The terminal operation, `toList`, is used to actually compute values and return them all in a list.

> On a 2017 MacBook Pro, even this nonoptimized algorithm generated 10,000 prime numbers in just under 50 milliseconds. Computers are fast now. FYI, the 10,000th prime is 104,729.

Another way of truncating an infinite sequence is to use a generation function that eventually returns null. Instead of asking for the first `N` primes, say you want all the prime numbers less than a particular limit. The `primesLessThan` function is shown in Example 6-8.

Example 6-8. Primes less than a given value (version 1)

```kotlin
fun primesLessThan(max: Int): List<Int> =
    generateSequence(2) { n -> if (n < max) nextPrime(n) else null }
        .toList().dropLast(1)
```

The function used in the `generateSequence` call in this case checks whether the current value is less than the supplied limit. If so, it computes the next prime. Otherwise, it returns `null`, and returning `null` terminates the sequence.

Of course, there's no way to know whether the next prime will be greater than the limit, so this function actually produces a list that includes the first prime above the limit. The `dropLast` function then truncates the resulting list before returning it.

Most of the time, there's an easier way to solve the same problem. In this case, rather than cause the generating function to return `null`, it's arguably easier to use `takeWhile` on the sequence, as in Example 6-9.

Example 6-9. Primes less than a given value (version 2)

```kotlin
fun primesLessThan(max: Int): List<Int> =
    generateSequence(2, ::nextPrime)
        .takeWhile { it < max }
        .toList()
```

The `takeWhile` function pulls values from the sequence as long as the supplied predicate returns `true`.

Either of these approaches work, so which to use is a matter of personal preference.

6.4 Yielding from a Sequence

Problem

You want to produce values in a sequence, but at specified intervals.

Solution

Use the `sequence` function along with the `yield` suspend function.

Discussion

Another function associated with sequences is `sequence`, which has the signature shown in Example 6-10.

Example 6-10. The signature of the `sequence` function

```
fun <T> sequence(
    block: suspend SequenceScope<T>.() -> Unit
): Sequence<T>
```

The `sequence` function produces a sequence by evaluating the given block. The block is a lambda function of no arguments that returns `void`, acting on the receiver of type `SequenceScope`.

That all sounds complicated until you see how it is used. Normally, to generate a sequence, you produce one from existing data using `sequenceOf`, you transform a collection into a sequence using `asSequence`, or you produce values with a function supplied to `generateSequence`. In this case, however, you supply a lambda that produces a value, which you yield whenever you like.

A good example is generating Fibonacci numbers, as shown in Example 6-11, based on an example from the library documentation.

Example 6-11. Generating Fibonacci numbers as a sequence

```
fun fibonacciSequence() = sequence {
    var terms = Pair(0, 1)

    while (true) {
        yield(terms.first)
        terms = terms.second to terms.first + terms.second
    }
}
```

The lambda provided to the `sequence` function starts off with a `Pair` containing the first two Fibonacci numbers, 0 and 1. Then it uses an infinite loop to produce the subsequent values. Each time a new element is produced, the `yield` function returns the `first` element of the resulting pair.

The `yield` function is one of two similar functions that are part of `SequenceScope`, the receiver of the lambda provided to the `sequence` operation. The signatures of both `yield` and `yieldAll` are shown in Example 6-12, along with their overloaded versions.

Example 6-12. The `yield` and `yieldAll` functions from `SequenceScope`

```
abstract suspend fun yield(value: T)

abstract suspend fun yieldAll(iterator: Iterator<T>)
suspend fun yieldAll(elements: Iterable<T>)
suspend fun yieldAll(sequence: Sequence<T>)
```

The job of the `yield` function is to provide a value to an iterator and suspend until the next value is requested. Therefore, in the sequence generated by the `suspend` function, `yield` is used to output individual values. The fact that `yield` is a `suspend` function means it plays nicely with coroutines. In other words, the Kotlin runtime can provide a value and then put the current coroutine on hold until the next value is requested. That's why the infinite loop—the `while(true)` loop in Example 6-11—provides values one by one when invoked by the `take` operation in Example 6-13.

Example 6-13. Pulling values from the `sequence` operation

```
@Test
fun `first 10 Fibonacci numbers from sequence`() {
    val fibs = fibonacciSequence()
        .take(10)
        .toList()

    assertEquals(listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34), fibs)
}
```

As you might expect, `yieldAll` yields several values to the iterator. The example given in the Kotlin documentation is shown in Example 6-14.

Example 6-14. The `yieldAll` function inside a `sequence`

```
vale sequence = sequence {
    val start = 0
    yield(start)                          ❶
    yield(1..5 step 2)                    ❷
    yield(generateSequence(8) { it * 3 }) ❸
}
```

❶ Yields a single value (0)

❷ Yields an iterable over the range (1, 3, 5)

❸ Yields an infinite sequence that starts at 8 and multiplies each value by 3

The result of this code is the sequence 0, 1, 3, 5, 8, 24, 72…. When the `sequence` is accessed with a `take` function, it returns as many elements as are requested, using the given pattern.

The combination of `yield` and `yieldAll` inside `suspend` makes it easy to customize a sequence to any desired set of generated values.

See Also

The `suspend` keyword is covered in more detail in the section on coroutines in Chapter 13.

CHAPTER 7

Scope Functions

The Kotlin standard library contains several functions whose purpose is to execute a block of code in the context of an object. Specifically, this chapter discusses the scope functions `let`, `run`, `apply`, and `also`.

7.1 Initializing Objects After Construction with apply

Problem

You need to initialize an object before using it, beyond what you can do by supplying constructor arguments.

Solution

Use the `apply` function.

Discussion

Kotlin has several *scoping functions* that you can apply to objects. The `apply` function is an extension function that sends `this` as an argument and returns it as well. Example 7-1 shows the definition of `apply`.

Example 7-1. Definition of the `apply` function

```
inline fun <T> T.apply(block: T.() -> Unit): T
```

The `apply` function is thus an extension function on any generic type `T`, which calls the specified block with `this` as its receiver and returns `this` when it completes.

As a practical example, consider the problem of saving an object to a relational database by using the Spring framework. Spring provides a class called SimpleJdbcInsert, based on JdbcTemplate, which removes the boilerplate from normal JDBC code in Java.

Say we have an entity called Officer that maps to a database table called OFFICERS. Writing the SQL INSERT statement for such a class is straightforward, except for one complication: if the primary key is generated by the database during the save, then the supplied object needs to be updated with the new key. For this purpose, the SimpleJdbcInsert class has a convenient method called executeAndReturnKey, which takes a map of column names to values and returns the generated value.

Using the apply function, the save function can receive an instance to be saved and update it with the new key all in one statement, as in Example 7-2.

Example 7-2. Inserting a domain object and updating the generated key

```
@Repository
class JdbcOfficerDAO(private val jdbcTemplate: JdbcTemplate) {

    private val insertOfficer = SimpleJdbcInsert(jdbcTemplate)
            .withTableName("OFFICERS")
            .usingGeneratedKeyColumns("id")

    fun save(officer: Officer) =
        officer.apply {
            id = insertOfficer.executeAndReturnKey(
                    mapOf("rank" to rank,
                          "first_name" to first,
                          "last_name" to last))
            }

// ...

}
```

The Officer instance is passed into the apply block as this, so it can be used to access the properties rank, first, and last. The id property of the officer is updated inside the apply block, and the officer instance is returned. Additional initialization could be chained to this block if necessary or desired.

The apply block is useful if the result needs to be the context object (the officer in this example). It is most commonly used to do additional configuration of objects that have already been instantiated.

7.2 Using also for Side Effects

Problem

You want to print a message or other side effect without interrupting the flow of your code.

Solution

Use the `also` function to perform the action.

Discussion

The function `also` is an extension function in the standard library, whose implementation is shown in Example 7-3.

Example 7-3. The extension function `also`

```
public inline fun <T> T.also(
    block: (T) -> Unit
): T
```

As the code shows, `also` is added to any generic type `T`, which it returns after executing the block argument. It is most commonly used to chain a function call onto an object, as in Example 7-4.

Example 7-4. Printing and logging with `also`

```
val book = createBook()
    .also { println(it) }
    .also { Logger.getAnonymousLogger().info(it.toString()) }
```

Inside the block, the object is referenced as `it`.

Because `also` returns the context object, it's easy to chain additional calls together, as shown here, where the book was first printed to the console and then logged somewhere.

While it's useful to see that you can chain multiple `also` calls together, the function is more typically added as part of a series of business logic calls. For example, consider a test of a geocoder service, given by Example 7-5.

Example 7-5. Testing a geocoder service

```kotlin
class Site(val name: String,
           val latitude: Double,
           val longitude: Double)

// ... inside test class ...

@Test
fun `lat,lng of Boston, MA`() = service.getLatLng("Boston", "MA")
    .also { logger.info(it.toString()) }  ❶
    .run {
        assertThat(latitude, `is`(closeTo(42.36, 0.01)))
        assertThat(longitude, `is`(closeTo(-71.06, 0.01)))
    }
```

❶ Logging as a side effect

This test could be organized in many ways, but using `also` in this way implies that the point of this code is to run the tests, but *also* to print the site. Note that using the scope functions converts the entire test into a single expression, allowing for the shorter syntax.

The `also` call has to come before the `run` call in the test, because `run` returns the value of the lambda rather than the context object.

Incidentally, although you could replace the `run` call with `apply`, JUnit tests are supposed to return `Unit`. The `run` call in Example 7-5 does that (because the assertions don't return anything), while `apply` would return the context object.

See Also

Recipe 7.1 discusses the `apply` function.

7.3 Using the let Function and Elvis

Problem

You want to execute a block of code only on a non-null reference, but return a default otherwise.

Solution

Use the `let` scope function with a safe call, combined with the Elvis operator.

Discussion

The `let` function is an extension function on any generic type `T`, whose implementation in the standard library is given by Example 7-6.

Example 7-6. Implementation of `let` in the standard library

```
public inline fun <T, R> T.let(
    block: (T) -> R
): R
```

The key fact to remember about `let` is that it returns the result of the block, rather than the context object. It therefore acts like a transformation of the context object, sort of like a `map` for objects. Say you want to take a string and capitalize it, but require special handling for empty or blank strings, as in Example 7-7.

Example 7-7. Capitalizing a string with special cases

```
fun processString(str: String) =
    str.let {
        when {
            it.isEmpty() -> "Empty"
            it.isBlank() -> "Blank"
            else -> it.capitalize()
        }
    }
```

Normally, you would just call the `capitalize` function, but on empty or blank strings this wouldn't give back anything useful. The `let` function allows you to wrap the `when` conditional inside a block that handles all the required cases, and returns the "transformed" string.

This really becomes interesting, however, when the argument is nullable, as in Example 7-8.

Example 7-8. Same process, but with a nullable string

```
fun processNullableString(str: String?) =
    str?.let {          ❶
        when {
            it.isEmpty() -> "Empty"
            it.isBlank() -> "Blank"
            else -> it.capitalize()
        }
    } ?: "Null"         ❷
```

❶ Safe call with a `let`

❷ Elvis operator to handle the null case

The return type on both functions is `String`, which is inferred from the execution.

In this case, the combination of the safe call operator `?.`, the `let` function, and the Elvis operator `?:` combine to handle all cases easily. This is a common idiom in Kotlin, as it lets (sorry) you handle both the null and non-null cases easily.

Many Java APIs (like Spring's `RestTemplate` or `WebClient`) return nulls when there is no result, and the combination of a safe call, a `let` block, and an Elvis operator is an effective means of handling them.

See Also

The `also` block is discussed in Recipe 7.2. Using `let` as a replacement for temporary variables is shown in Recipe 7.4.

7.4 Using let with a Temporary Variable

Problem

You want to process the result of a calculation without needing to assign the result to a temporary variable.

Solution

Chain a `let` call to the calculation and process the result in the supplied lambda or function reference.

Discussion

The documentation pages for scope functions at the Kotlin website show an interesting use case for the `let` function. Their example (repeated in Example 7-9) creates a mutable list of strings, maps them to their lengths, and filters the result.

Example 7-9. `let` example from online docs, before refactoring

```
// Before
val numbers = mutableListOf("one", "two", "three", "four", "five")
val resultList = numbers.map { it.length }.filter { it > 3 }
println(resultList)  ❶
```

❶ Assigns calculation to temp variable for printing

After refactoring to use a `let` block, the code looks like Example 7-10.

Example 7-10. After refactoring to use `let`

```
// After
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let {
    println(it)
    // and more function calls if needed
}
```

The idea is that rather than assign the result to a temporary variable, the chained `let` call uses the result as its context variable, so it can be printed (or more) in the provided block. If all that is required is to print the result, this can even be reduced further, to the form in Example 7-11.

Example 7-11. Using a function reference in the `let` block

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let(::println)
```

As a slightly more interesting example, consider a class that accesses a remote service at Open Notify that returns the number of astronauts in space, as described in Recipe 11.6 later. The service returns JavaScript Object Notation (JSON) data and transforms the result into instances of classes that you'll see again in Example 11-17:

```
data class AstroResult(
    val message: String,
    val number: Number,
    val people: List<Assignment>
)

data class Assignment(
    val craft: String,
    val name: String
)
```

Example 7-12 uses the extension method `URL.readText` and Google's Gson library to convert the received JSON into an instance of `AstroResult`.

Example 7-12. Printing the names of astronauts currently in space

```
Gson().fromJson(
    URL("http://api.open-notify.org/astros.json").readText(),
        AstroResult::class.java
    ).people.map { it.name }.let(::println)  ❶
```

❶ Uses `let` (or `also`) to print the `List<String>`

In this case, the basic code in the `Gson().fromJson` call converts the JSON data into an instance of `AstroResult`. The `map` function then transforms the `Assignment` instances into a list of strings representing the astronaut names.

As of August 2019, the output from this program is (all on one line):

```
    [Alexey Ovchinin, Nick Hague, Christina Koch,
        Alexander Skvortsov, Luca Parmitano, Andrew Morgan]
```

In this case, the `let` in Example 7-12 could be replaced with an `also`. The difference is that `let` returns the result of the block (`Unit` in the case of `println`) while `also` would return the context object (the `List<String>`). Neither is used after the print, so the difference in this case doesn't matter. It might be more idiomatic to use `also`, since that is typically used for side effects like printing. Either way works, though.

See Also

See Recipe 7.3 for how to use `let` with a safe call and Elvis operator in the case of nullable values. The `also` function is discussed in Recipe 7.2.

CHAPTER 8

Kotlin Delegates

This chapter talks about delegates in Kotlin. You'll learn how to use delegates in the standard library, including `lazy`, `observable`, `vetoable`, and `notNull`, as well as create your own. *Class delegates* let you replace inheritance with composition, and *property delegates* replace the getters and setters for a property with those from another class.

In addition to the basic demonstrations, this chapter also shows how some of the standard delegates are implemented in the library, as examples of good idiomatic usage.

8.1 Implementing Composition by Delegation

Problem

You want to create a class that contains instances of other classes and delegate behavior to them.

Solution

Create interfaces that contain the delegation methods, implement them in classes, and build the wrapper class out of them using the keyword `by`.

Discussion

Modern object-oriented design tends to favor composition rather than inheritance[1] as a way to add functionality without strong coupling. In Kotlin, the keyword `by` allows a class to expose all the public functions in a contained object through the container.

For example, a smartphone contains both a phone and a camera, among other components. If you think of the smartphone as a *wrapper* object, and the internal phone and camera as *contained* objects, the goal is to write the smartphone class so that its functions invoke the corresponding ones in the contained instances.

To do this in Kotlin, you need to create interfaces for the exposed methods in the contained objects. Consider, therefore, the `Dialable` and `Snappable` interfaces that are implemented by `Phone` and `Camera` in Example 8-1.

Example 8-1. Interfaces and classes for contained objects

```
interface Dialable {
    fun dial(number: String): String
}

class Phone : Dialable {
    override fun dial(number: String) =
        "Dialing $number..."
}

interface Snappable {
    fun takePicture(): String
}

class Camera : Snappable {
    override fun takePicture() =
        "Taking picture..."
}
```

Now a `SmartPhone` class can be defined that instantiates a phone and a camera instance in the constructor and delegates all the public functions to them, as in Example 8-2.

Example 8-2. The `SmartPhone` delegates to the contained instances

```
class SmartPhone(
    private val phone: Dialable = Phone(),
```

1 Really bad joke alert: Beethoven's parents wouldn't give him the family money unless he gave up writing music, yet he still chose composition over inheritance. (Rim shot. Also, to be honest, not true.)

```
    private val camera: Snappable = Camera()
) : Dialable by phone, Snappable by camera      ❶
```

❶ Delegation using the by keyword

Now instantiating a `SmartPhone` allows you to invoke all the methods in `Phone` or `Camera`, as the test cases in Example 8-3 show.

Example 8-3. Tests for the `SmartPhone`

```
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.Assertions.*

class SmartPhoneTest {
    private val smartPhone: SmartPhone = SmartPhone()    ❶

    @Test
    fun `Dialing delegates to internal phone`() {
        assertEquals("Dialing 555-1234...",
            smartPhone.dial("555-1234"))                 ❷
    }

    @Test
    fun `Taking picture delegates to internal camera`() {
        assertEquals("Taking picture...",
            smartPhone.takePicture())                    ❷
    }
}
```

❶ Instantiates `SmartPhone` with no arguments

❷ Invokes delegated functions

The contained objects themselves (the instances of phone and camera) are not exposed through the smartphone; only their public functions are. The `Phone` and `Camera` classes could have many other functions, but only the ones declared in the corresponding interfaces `Dialable` and `Snappable` are available. The extra work of defining the interfaces seems like overkill, but it does keep the relationships clean.

If you play the typical game in IntelliJ of showing the Kotlin bytecode and then decompiling it, the corresponding Java code includes the snippet in Example 8-4.

Example 8-4. Part of the decompiled bytecode from `SmartPhone`

```
public final class SmartPhone implements Dialable, Snappable {
   private final Dialable phone;          ❶
   private final Snappable camera;        ❶
```

```
    public SmartPhone(@NotNull Dialable phone, @NotNull Snappable camera) {
        // ...
        this.phone = phone;
        this.camera = camera;
    }

    @NotNull
    public String dial(@NotNull String number) {
        return this.phone.dial(number);      ❷
    }

    @NotNull
    public String takePicture() {
        return this.camera.takePicture();    ❷
    }

    // ...
}
```

❶ Fields of interface type

❷ Delegation methods

Internally, the `SmartPhone` class defines the delegated properties as interface types. The corresponding class instances are supplied in the constructor. Then the delegation methods invoke the corresponding methods on the fields.

See Also

Delegated properties are examined in Recipe 8.6.

8.2 Using the lazy Delegate

Problem

You want to wait to initialize a property until it is needed.

Solution

Use the `lazy` delegate in the standard library.

Discussion

Kotlin uses the `by` keyword on properties to imply that its getter and setter are implemented by a different object, called a *delegate*. Several delegate functions are in the standard library. One of the most popular is called `lazy`.

To use it, provide a lambda initializer of the form `() -> T` whose purpose is to compute the value when it is first accessed, as shown in Example 8-5.

Example 8-5. Signatures of `lazy` function

```
fun <T> lazy(initializer: () -> T): Lazy<T>                  ❶

fun <T> lazy(                                                ❷
    mode: LazyThreadSafetyMode,
    initializer: () -> T
): Lazy<T>

fun <T> lazy(lock: Any?, initializer: () -> T): Lazy<T>      ❸
```

❶ Default, synchronized on itself

❷ Specifies how the instance synchronizes initialization among multiple threads

❸ Uses the provided object for the synchronization lock

The versions without the `mode` property default to `LazyThreadSafetyMode.SYNCHRONIZED`. If the initialization lambda throws an exception, it will attempt to reinitialize the value at the next access. See Example 8-6 for a trivial example.

Example 8-6. Waiting to initialize a property until it is first accessed

```
val ultimateAnswer: Int by lazy {
    println("computing the answer")
    42
}
```

The idea is that the value of `ultimateAnswer` is not computed until it is first accessed, at which point the lambda expression is evaluated. In the implementation, `lazy` is a function that takes a lambda and returns an instance of `Lazy<Int>` that will execute the lambda when the property is first accessed.

So the following code prints the statement "computing the answer" only once:

```
    println(ultimateAnswer)
    println(ultimateAnswer)
```

The first call to `ultimateAnswer` executes the lambda and returns the value 42, which is then saved in the variable. Internally, Kotlin generates a special property called `myAnswer$delegate` of type `Lazy`, which is used to cache the value.

The argument of type `LazyThreadSafetyMode` takes an enum whose values can be as follows:

`SYNCHRONIZED`
: Locks are used to ensure that only a single thread can initialize the `Lazy` instance.

`PUBLICATION`
: Initializer function can be called several times, but only the first returned value will be used.

`NONE`
: No locks are used.

If an object is provided as the `lock` argument, the delegate synchronizes on that object when computing the value. Otherwise, it synchronizes on itself.

The `lazy` delegate is appropriate when the instantiation involves more complex objects, but the principles are the same in any case.

As an aside, the implementation of `lazy` in the standard library does not follow the same pattern as the rest of the delegates. The `lazy` function is a top-level function, while most of the rest are part of the `Delegates` instance discussed in the other recipes in this chapter.

8.3 Ensuring That a Value Is Not Null

Problem

You want to throw an exception if a value has not been initialized before first access.

Solution

Use the `notNull` function to provide a delegate that throws an exception if the value has not been set.

Discussion

Normally, a property in a Kotlin class is initialized during construction. One way to delay this is to use the `notNull` function, which provides a delegate that throws an exception if the property is used before it is first accessed.

Example 8-7 declares a property called `shouldNotBeNull` that must be initialized somewhere before it is used.

Example 8-7. Require initialization before access, without specifying how

```kotlin
var shouldNotBeNull: String by Delegates.notNull<String>()
```

The test cases in Example 8-8 show that if you try to access the property before giving it a value, Kotlin will throw an `IllegalStateException`.

Example 8-8. Checking the behavior of the `notNull` delegate

```
@Test
fun `uninitialized value throws exception`() {
    assertThrows<IllegalStateException> { shouldNotBeNull }
}

@Test
fun `initialize value then retrieve it`() {
    shouldNotBeNull = "Hello, World!"
    assertDoesNotThrow { shouldNotBeNull }
    assertEquals("Hello, World!", shouldNotBeNull)
}
```

While this behavior is straightforward enough, the interesting part comes when you look at the implementation in *Delegates.kt* in the standard library. An abbreviated version of that file is shown in Example 8-9.

Example 8-9. Implementation in standard library

```
object Delegates {                                                    ❶
    fun <T : Any> notNull(): ReadWriteProperty<Any?, T> = NotNullVar() ❷

    // ... additional functions discussed in other recipes ...
}

private class NotNullVar<T : Any>() : ReadWriteProperty<Any?, T> {    ❸
    private var value: T? = null

    override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return value ?: throw IllegalStateException(
            "Property ${property.name} should be initialized before get.")
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        this.value = value
    }
}
```

❶ `Delegates` is a singleton (`object` rather than `class`)

❷ Factory method that instantiates `NotNullVar` class

❸ Private class that implements `ReadWriteProperty`

The keyword `object` is used to define a singleton instance of `Delegates`, so the included `notNull` function acts as though it were `static` in Java. This factory method then instantiates the private class `NotNullVar`, which implements the `ReadWriteProperty` interface.

As discussed in Recipe 8.6, when you write your own custom delegates, you don't have to implement this interface (or the associated interface `ReadOnlyProperty` used when the property cannot be changed), but you do need to include the two methods shown. In the case of `NotNullVar`, the `setValue` function simply stores the supplied value, while the `getValue` function checks that it is not null and either returns it or throws an `IllegalStateException`.[2]

This combination of a singleton class, a factory method, and a private implementation class is a common idiom in Kotlin. If you provide your own custom delegate, consider following this pattern.

8.4 Using the observable and vetoable Delegates

Problem

You want to intercept changes to a property and optionally veto them.

Solution

Use the `observable` function to detect changes, and the `vetoable` function with a lambda to decide whether to implement them.

Discussion

As with Recipe 8.3, the `observable` and `vetoable` functions in the `Delegates` object in the standard library are easy to use, but the implementations demonstrate a good pattern for writing your own.

Before getting into that, however, consider how the functions are used. In the documentation, the signatures for the two functions are given by Example 8-10.

Example 8-10. Signatures for the `observable` and `vetoable` functions

```
fun <T> observable(
    initialValue: T,
    onChange: (property: KProperty<*>, oldValue: T, newValue: T) -> Unit
```

2 Bad joke alert: The author lives in eastern Connecticut, so there `IllegalStateException` means New York. (Sorry again.)

```
): ReadWriteProperty<Any?, T>

fun <T> vetoable(
    initialValue: T,
    onChange: (property: KProperty<*>, oldValue: T, newValue: T) -> Boolean
): ReadWriteProperty<Any?, T>
```

Both factory functions take an initial value of type `T` and a lambda, and return an instance of a class that implements the `ReadWriteProperty` interface. Using them is straightforward, as the examples in Example 8-11 show.

Example 8-11. Using the `observable` and `vetoable` functions

```
var watched: Int by Delegates.observable(1) { prop, old, new ->
    println("${prop.name} changed from $old to $new")
}

var checked: Int by Delegates.vetoable(0) { prop, old, new ->
    println("Trying to change ${prop.name} from $old to $new")
    new >= 0
}
```

The `watched` variable is an `Int` that is initialized to 1, and whenever it is changed, a message is printed showing the old and new values. The `checked` variable is also an `Int`, initialized to 0, but this time only non-negative values are allowed. The lambda argument returns `true` only if the new value is greater than or equal to 0.

A test for the `watched` variable in Example 8-12 shows the value changes as expected.

Example 8-12. Test for the `watched` variable

```
@Test
fun `watched variable prints old and new values`() {
    assertEquals(1, watched)
    watched *= 2
    assertEquals(2, watched)
    watched *= 2
    assertEquals(4, watched)
}
```

This test prints to the console:

```
watched changed from 1 to 2
watched changed from 2 to 4
```

For the vetoable `checked` variable, the test in Example 8-13 shows that only values greater than or equal to 0 are accepted.

Example 8-13. Testing the vetoable changes for `checked`

```
@Test
fun `veto values less than zero`() {
    assertAll(
        { assertEquals(0, checked) },
        { checked = 42; assertEquals(42, checked) },
        { checked = -1; assertEquals(42, checked) },
        { checked = 17; assertEquals(17, checked) }
    )
}
```

Changing the value of `checked` to 42 or 17 is fine, but –1 is rejected.

Using either function is pretty straightforward, but again the really interesting part is the way they are implemented. As with `notNull`, both `observable` and `vetoable` are factory functions in the singleton `Delegates` object, as in Example 8-14.

Example 8-14. Factory functions in `Delegates`

```
object Delegates {
    // ... others ...

    inline fun <T> observable(initialValue: T,
        crossinline onChange: (property: KProperty<*>,
            oldValue: T, newValue: T) -> Unit): ReadWriteProperty<Any?, T> =
        object : ObservableProperty<T>(initialValue) {
            override fun afterChange(property: KProperty<*>,
            oldValue: T, newValue: T) = onChange(property, oldValue, newValue)
        }

    inline fun <T> vetoable(initialValue: T,
        crossinline onChange: (property: KProperty<*>,
            oldValue: T, newValue: T) -> Boolean): ReadWriteProperty<Any?, T> =
        object : ObservableProperty<T>(initialValue) {
            override fun beforeChange(property: KProperty<*>,
            oldValue: T, newValue: T): Boolean =
                onChange(property, oldValue, newValue)
        }
}
```

Those look complicated, to say the least. The first thing to notice is that both functions return an object of type `ObservableProperty`. That class is shown in Example 8-15.

Example 8-15. The `ObservableProperty` class used to provide a delegate

```
abstract class ObservableProperty<T>(initialValue: T) : ReadWriteProperty<Any?, T> {
    private var value = initialValue

    protected open fun beforeChange(property: KProperty<*>,
        oldValue: T, newValue: T): Boolean = true

    protected open fun afterChange(property: KProperty<*>,
        oldValue: T, newValue: T): Unit {}

    override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return value
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        val oldValue = this.value
        if (!beforeChange(property, oldValue, value)) {
            return
        }
        this.value = value
        afterChange(property, oldValue, value)
    }
}
```

The class stores a property of any generic type `T` and implements the `ReadWriteProperty` interface. That means it needs to provide the `getValue` and `setValue` functions with the signatures shown. In this case, `getValue` just returns the property.

Things get interesting in the `setValue` function. This function stores the current value and then invokes the `beforeChange` method. If that function returns `true` (and the default is `true`), then the property is changed and the `afterChange` function is invoked, which defaults to doing nothing.

This is an abstract class, whose `open` functions are `beforeChange` and `afterChange`, both of which are marked `protected`. They both have default implementations, but subclasses are free to override those functions.

That's where the implementations in Example 8-14 come in. The `observable` function makes an object that extends `ObservableProperty` and overrides the `afterChange` function to do whatever the supplied `onChange` lambda specifies. Since it does not override `beforeChanged`, that function simply returns `true`, ensuring that the property will be updated.

The `vetoable` function, on the other hand, also creates an object from a class that extends `ObservableProperty`, but in this case only the `beforeChanged` function is overridden. The lambda implementation of that is supplied as an argument, and it must return a boolean, which determines whether the property will be changed.

Again, the work done to create the `ObservableProperty` class to implement the `ReadWriteProperty` interface but with additional life cycle methods makes it trivially easy to implement both the `observable` and `vetoable` functions.

inline Versus crossinline

The `inline` keyword tells the compiler to avoid creating a completely new object just to call a function, but rather to replace the call site with the actual source code.

Sometimes `inline` functions are passed lambdas as parameters that need to be executed from another context, such as a local object or nested function. Such "nonlocal" control flow is not allowed in lambdas. In the examples shown, the `onChange` lambda is executed in relation to the `observable` or `vetoable` functions rather than the class extending `ObservableProperty`, so the `crossinline` modifier is necessary.

The combination of factory functions inside a singleton object that configure an instance of a delegate class is a powerful one.

8.5 Supplying Maps as Delegates

Problem

You want to supply a map of values to initialize an object.

Solution

Kotlin maps already implement the `getValue` and `setValue` functions necessary to be a delegate.

Discussion

If the values you need to initialize an object are in a map, you can automatically delegate the class properties to that map. For example, say you have a `Project` class as in Example 8-16.

Example 8-16. A `Project` data class

```
data class Project(val map: MutableMap<String, Any?>) {
    val name: String by map          ❶
    var priority: Int by map          ❶
    var completed: Boolean by map    ❶
}
```

❶ Delegation to the `map` argument

In this case, the `Project` constructor takes a `MutableMap` as an argument, and all the properties are initialized from its keys. Creating an instance of the `Project` type requires a map, as in Example 8-17.

Example 8-17. Creating a `Project` instance from a map

```
@Test
fun `use map delegate for Project`() {
    val project = Project(
        mutableMapOf(
            "name" to "Learn Kotlin",
            "priority" to 5,
            "completed" to true))

    assertAll(
        { assertEquals("Learn Kotlin", project.name) },
        { assertEquals(5, project.priority) },
        { assertTrue(project.completed) }
    )
}
```

This works because `MutableMap` has extension functions `setValue` and `getValue` with the proper signatures, as required in order to be a `ReadWriteProperty` delegate.

You might wonder, however, why the extra layer of indirection was required. In other words, why not just make the properties part of the constructor, rather than using a map? The documentation suggests that this mechanism arises in applications "like parsing JSON or doing other dynamic things."

Fair enough. The test in Example 8-18 assumes that the required properties are in a JSON string, hardcoded here for simplicity. Then Google's Gson library is used to parse the string, and the resulting map is used to create a `Project` instance.

Example 8-18. Parsing `Project` properties from a JSON string

```
private fun getMapFromJSON() =                              ❶
    Gson().fromJson<MutableMap<String, Any?>>(
        """{ "name":"Learn Kotlin", "priority":5, "completed":true}""",
        MutableMap::class.java)

@Test
fun `create project from map parsed from JSON string`() {
    val project = Project(getMapFromJSON())          // ❷
        assertAll(
            { assertEquals("Learn Kotlin", project.name) },
            { assertEquals(5, project.priority) },
            { assertTrue(project.completed) }
        )
    }
```

❶ Parses a map of properties from a JSON string

❷ Uses the map to instantiate a `Project` instance

The `fromJson` function from Gson takes a string and a type, so with Kotlin you can specify the generic type as shown. The resulting map is of the right type to provide values as a delegate.

8.6 Creating Your Own Delegates

Problem

You want properties of a given class to use getters and setters from another class.

Solution

Write your own property delegates by creating a class that implements either `ReadOnlyProperty` or `ReadWriteProperty`.

Discussion

Normally, a property of a class works with a backing field, but that's not required. Instead, the act of getting or setting a value can be delegated to another object. To create your own property delegate, you need to provide the functions from either the `ReadOnlyProperty` or the `ReadWriteProperty` interfaces.

The signatures for both interfaces are given in Example 8-19.

Example 8-19. The `ReadOnlyProperty` and `ReadWriteProperty` interfaces

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}
```

Interestingly enough, you don't need to implement either of these interfaces to make a delegate. Simply having the `getValue` and `setValue` functions with the signatures shown is enough.

A trivial example of a delegate is given in the standard documentation for delegated properties, which includes a class called `Delegate`; see Example 8-20.

Example 8-20. The `Delegate` class from the standard documentation

```kotlin
class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

Using this delegate involves creating a class or variable that delegates to it and then getting or setting that variable, as in Example 8-21.

Example 8-21. Using `Delegate`

```kotlin
class Example {
    var p: String by Delegate()
}

fun main() {
    val e = Example()
    println(e.p)
    e.p = "NEW"
}
```

This prints the following:

```
delegates.Example@4c98385c, thank you for delegating 'p' to me!
NEW has been assigned to 'p' in delegates.Example@4c98385c.
```

Creating a class property wasn't strictly necessary. As of Kotlin 1.1, you can delegate a local variable as well.

The standard library includes several delegates. Recipe 8.3 shows the code for the `notNull` function, which instantiates the private class `NotNullVar` shown in Example 8-9, for example.

As a completely different set of examples, the Gradle build tool now provides a Kotlin DSL that lets you interact with containers via delegated properties. Gradle has two main sources of properties. One is the set of properties associated with the project itself (an instance of the `org.gradle.api.Project` class), and the others are called `extra` properties that can be used throughout a project.

Say you have a build file called *build.gradle.kts*. Then both types of properties can be created and accessed as in Example 8-22.

Example 8-22. Creating and accessing project and extra properties in the Gradle Kotlin DSL

```
val myProperty: String by project                          ❶
val myNullableProperty: String? by project                 ❷

val myNewProperty by extra("initial value")                ❸
val myOtherNewProperty by extra { "lazy initial value" }   ❹
```

❶ Makes the project property called `myProperty` available

❷ Makes a nullable property available

❸ Creates and initializes a new extra property called `myNewProperty`

❹ Creates a property that is initialized on first access

Project properties can be set on the command line, using the `-PmyProperty=value` syntax, or they can be set in a *gradle.properties file*. Extra properties defined as shown are also initialized either with the given argument or with a lambda that is evaluated on first access.

Creating property delegates is reasonably straightforward, but as the rest of this chapter shows, you're more likely to use existing ones, either from the library or supplied by third parties like Gradle.

CHAPTER 9
Testing

9.1 Setting the Test Class Life Cycle

Problem

You want to instantiate a JUnit 5 test only once per class instance, rather than the default once per test function.

Solution

Use the `@TestInstance` annotation, or set the life cycle default property in the *junit-platform.properties* file.

Discussion

This recipe, as with many of the recipes in this chapter, is based on a blog post and presentation entitled, "Best Practices for Unit Testing in Kotlin," by Philipp Hauer.

JUnit 4 by default creates a new instance of the test class for each test method. This ensures that the attributes of the test class are reinitialized each time, which makes the tests themselves independent. The downside is that the initialization code is executed again and again for each test.

To avoid that in Java, any properties of the class can be marked `static`, and any initialization code can be put in a static method that is annotated with `@BeforeClass`, which will be executed only once.

As an example, consider the following JUnit 4 test for `java.util.List`, shown in Example 9-1.

Example 9-1. JUnit 4 test for a list

```
public class JUnit4ListTests {
    private static List<String> strings =                      ❶
            Arrays.asList("this", "is", "a", "list", "of", "strings");

    private List<Integer> modifiable = new ArrayList<>();      ❷

    @BeforeClass                                               ❶
    public static void runBefore() {
        System.out.println("BeforeClass: " + strings);
    }

    @Before                                                    ❷
    public void initialize() {
        System.out.println("Before: " + modifiable);
        modifiable.add(3);
        modifiable.add(1);
        modifiable.add(4);
        modifiable.add(1);
        modifiable.add(5);
    }

    @Test
    public void test1() {
        // ...
    }

    @Test
    public void test2() {
        // ...
    }

    @Test
    public void test3() {
        // ...
    }

    @After                                                    ❷
    public void finish() {
        System.out.println("After: " + modifiable);
    }

    @AfterClass                                               ❶
    public static void runAfter() {
        System.out.println("AfterClass: " + strings);
    }
}
```

❶ Runs only once for the entire class

❷ Runs once per test method

The test class has two attributes, `strings` and `modifiable`. The `modifiable` list is structured to demonstrate the life cycle. It is initialized as an empty list where the attribute is declared, and then populated in the `@Before` method, `initialize`. This is to show that the list is empty each time that method is entered, and then populated. It is also printed in the `@After` method, `finish`, to show that it now contains the desired elements.

The `strings` list is also created and initialized each time, as part of the attribute declaration. To prevent this redundant work from being done before each test method, it is marked `static`. The `@BeforeClass` and `@AfterClass` life cycle methods are used to show that the `strings` collection is correctly populated.

To get the same behavior in Kotlin, you immediately encounter the problem that Kotlin does not have the `static` keyword. A simple port to Kotlin to get the same behavior would use the companion object, as shown in Example 9-2.

Example 9-2. JUnit 4 list tests in Kotlin (see JUnit 5 for better approach)

```
class JUnit4ListTests {
    companion object {          ❶
        @JvmStatic              ❷
        private val strings = listOf("this", "is", "a", "list", "of", "strings")

        @BeforeClass            ❶
        @JvmStatic              ❷
        fun runBefore() {
            println("BeforeClass: $strings")
        }

        @AfterClass             ❶
        @JvmStatic              ❷
        fun runAfter() {
            println("AfterClass: $strings")
        }
    }

    private val modifiable = ArrayList<Int>()

    @Before
    fun initialize() {
        println("Before: $modifiable")
        modifiable.add(3)
        modifiable.add(1)
        modifiable.add(4)
```

```
            modifiable.add(1)
            modifiable.add(5)
        }

        @Test
        fun test1() {
            // ...
        }

        @Test
        fun test2() {
            // ...
        }

        @Test
        fun test3() {
            // ...
        }

        @After
        fun finish() {
            println("After: $modifiable")
        }
    }
```

❶ Uses companion object to run once per class

❷ Ensures generated Java bytecodes use `static` modifier

The companion object is used to ensure that the `strings` collection is instantiated and populated only once for the entire class. The `@BeforeClass` and `@AfterClass` methods are also used inside the companion object, for the same purpose. Note that if the instantiation of the list was also done in the `initialize` method, the `strings` attribute would need to be declared with `lateinit` and as a `var` type rather than `val`:

```
class JUnit4ListTests {
    companion object {
        @JvmStatic
        private lateinit var strings

        @BeforeClass
        @JvmStatic
        fun runBefore() {
            strings = listOf("this", "is", "a", "list", "of", "strings")
        }
// ...
}
```

If this were necessary (i.e., if we were testing a more complex object than a simple list, which needed additional configuration), then the use of `var` makes this even less idiomatic Kotlin.

Fortunately, JUnit 5 provides a simpler approach. JUnit 5 allows you to specify the life cycle of the test class itself, using the `@TestInstance` annotation. A better restatement of the list tests is shown in Example 9-3.

Example 9-3. JUnit 5 list tests in Kotlin (preferred)

```kotlin
import org.junit.jupiter.api.*
import org.junit.jupiter.api.Assertions.assertEquals

@TestInstance(TestInstance.Lifecycle.PER_CLASS)        ❶
class JUnit5ListTests {
    private val strings =                               ❷
        listOf("this", "is", "a", "list", "of", "strings")

    private lateinit var modifiable : MutableList<Int> ❸

    @BeforeEach
    fun setUp() {
        modifiable = mutableListOf(3, 1, 4, 1, 5)       ❸
        println("Before: $modifiable")
    }

    @AfterEach
    fun finish() {
        println("After: $modifiable")
    }

    @Test
    fun test1() {
        // ...
    }

    @Test
    fun test2() {
        // ...
    }

    @Test
    fun test3() {
        // ...
    }
```

❶ Only one test class instance for all tests

❷ Instantiated and populated only once

❸ Reinitialized before each test

This is far more idiomatic. By setting the test instance life cycle to `PER_CLASS`, only one instance of the test is created regardless of how many test functions are included. That means the `strings` attribute can be created and populated in the normal way, using a `val` modifier as well.

A complication still arises if an attribute needs to be reinitialized between tests. The test class shows that you can still use `@BeforeEach` and `@AfterEach` as necessary, though in this case the attribute needs to use the `lateinit` and `var` modifiers. That is a quirk of this particular class, however, because we're using the `mutableList` function to instantiate and populate the list rather than instantiating it ourselves. For more complex (and more interesting) objects, you can instantiate them and then use the `apply` method to configure them, as shown in the next recipe.

JUnit 5 allows you to set the test life cycle for all tests in a properties file, rather than having to repeat the `@TestInstance` annotation on each test. If you create a file called *junit-platform.properties* in the classpath (normally in the *src/test/resources* folder), add the single line shown in Example 9-4.

Example 9-4. Setting the test life cycle for all tests in the project

```
junit.jupiter.testinstance.lifecycle.default = per_class
```

The only downside is that someone reading the test class has to know to check this file, because the default behavior in JUnit 5 is still to instantiate the class for each test function.

9.2 Using Data Classes for Tests

Problem

You want to check multiple properties of an object without bloating your code.

Solution

Create a data class that encapsulates all the desired properties.

Discussion

Data classes in Kotlin automatically include overrides for `equals`, `toString`, `hashCode`, `copy`, and component methods for destructuring. This makes them ideal for wrapping properties for tests.

Say you have a service that returns books based on ISBN numbers. The `Book` class is a data class given in Example 9-5.

Example 9-5. A `Book` data class

```
data class Book(
    val isbn: String,
    val title: String,
    val author: String,
    val published: LocalDate
)
```

For a given book, you could manually test it by checking all the properties, as in Example 9-6.

Example 9-6. Manually test all properties of a book (tedious)

```
@Test
internal fun `test book the hard way`() {
    val book = service.findBookById("1935182943")
    assertThat(book.isbn, `is`("1935182943"))
    assertThat(book.title, `is`("Making Java Groovy"))
    assertThat(book.author, `is`("Ken Kousen"))
    assertThat(book.published, `is`(LocalDate.of(2013, Month.SEPTEMBER, 30)))
}
```

This would work, but requires you to write out all the property assertions explicitly. The other problem is that if the first assertion fails, it will cause the whole test to fail, so some properties may not get checked at all. Fortunately, JUnit 5 adds the `assertAll` method that takes a `vararg` list of `Executable` instances, where `Executable` is a functional interface defined in Java that takes zero arguments and returns nothing. The advantage of the `assertAll` function is that it will execute all the `Executable` instances even if one or more fail.

Example 9-7 rewrites the previous example to take advantage of this capability.

Example 9-7. Using `assertAll` from JUnit 5 to test all properties

```
@Test
fun `use JUnit 5 assertAll`() {
    val book = service.findBookById("1935182943")
    assertAll("check all properties of a book",
        { assertThat(book.isbn, `is`("1935182943")) },
        { assertThat(book.title, `is`("Making Java Groovy")) },
        { assertThat(book.author, `is`("Ken Kousen")) },
        { assertThat(book.published,
`is`(LocalDate.of(2013, Month.SEPTEMBER, 30))) })
}
```

Note the use of Kotlin lambdas to represent instances of `Executable`. The provided lambdas match because they each take no arguments and the `assertThat` function returns `void`.

Still, you have to write all the individual tests, and that's annoying. Since Kotlin data classes already have an `equals` method implemented correctly, this process can be simplified, as in Example 9-8.

Example 9-8. Using the `Book` data class for testing

```
@Test
internal fun `use data class`() {
    val book = service.findBookById("1935182943")
    val expected = Book(isbn = "1935182943",
        title = "Making Java Groovy",
        author = "Ken Kousen",
        published = LocalDate.of(2013, Month.SEPTEMBER, 30))

    assertThat(book, `is`(expected)) ❶
}
```

❶ Single assertion does all the work

Now the assertion takes advantage of the data class override of the `equals` method.

If you have a collection of instances, you can take advantage of methods provided in the Hamcrest matchers to check all the elements, as in Example 9-9.

Example 9-9. Testing all returned books

```
@Test
internal fun `check all elements in list`() {
    val found = service.findAllBooksById(
        "1935182943", "1491947020", "149197317X")

    val expected = arrayOf(
        Book("1935182943", "Making Java Groovy",
            "Ken Kousen", LocalDate.parse("2013-09-30")),
        Book("1491947020", "Gradle Recipes for Android",
            "Ken Kousen", LocalDate.parse("2016-06-17")),
        Book("149197317X", "Modern Java Recipes",
            "Ken Kousen", LocalDate.parse("2017-08-26")))

    assertThat(found, arrayContainingInAnyOrder(*expected))
}
```

The Hamcrest method `arrayContainingInAnyOrder` takes a `vararg` list of individual elements, so this example uses the *spread* operator on the array, `*expected`, to expand the array into individual entries.

9.3 Using Helper Functions with Default Arguments

Problem

You want to rapidly create test objects.

Solution

Provide a helper function with default arguments, rather than use `copy` or specifying default constructor arguments where they aren't necessarily warranted.

Discussion

Creating test objects can be tedious. Kotlin allows you to specify default values for the arguments in the primary constructor of a class, but sometimes there are no obvious values. For example, consider the `Book` class shown in Example 9-5, repeated here for reference:

```
data class Book(
    val isbn: String,
    val title: String,
    val author: String,
    val published: LocalDate
)
```

Rather than modifying the class to give default values for each argument, add a factory function to produce a default, as in Example 9-10.

Example 9-10. Factory function for `Book`

```
fun createBook(
    isbn: String = "149197317X",
    title: String = "Modern Java Recipes",
    author: String = "Ken Kousen",
    published: LocalDate = LocalDate.parse("2017-08-26")
) = Book(isbn, title, author, published)
```

Using the factory function is as simple as Example 9-11.

Example 9-11. Creating books from the factory function

```
val modern_java_recipes = createBook()                      ❶
val making_java_groovy = createBook(isbn = "1935182943",    ❷
    title = "Making Java Groovy",
    published = LocalDate.parse("2013-09-30"))
```

❶ All defaults from factory

❷ Same author

The default arguments are being used only to create test data, so there's no need to add them to the domain class itself.

In principle, you could use the `copy` function provided on data classes to do the same thing, but extensive use of `copy` can be difficult to read, particularly on nested structures. The factory function in Example 9-12 shows that the function approach stays simple.

Example 9-12. A multiauthor book class and usage

```kotlin
data class MultiAuthorBook(
    val isbn: String,
    val title: String,
    val authors: List<String>,
    val published: LocalDate
)

fun createMultiAuthorBook(
    isbn: String = "9781617293290",
    title: String = "Kotlin in Action",
    authors: List<String> = listOf("Dimitry Jeremov",
                                   "Svetlana Isakova"),
    published: LocalDate = LocalDate.parse("2017-08-26")
) = MultiAuthorBook(isbn, title, authors, published)

val kotlin_in_action = createMultiAuthorBook()
```

If you put all the factory functions in a top-level utility class, you can reuse them in the tests.

9.4 Repeating JUnit 5 Tests with Different Data

Problem

You want to execute a JUnit 5 test with a given set of data values.

Solution

Use JUnit 5's parameterized tests and dynamic tests.

Discussion

Say you want to test a function by using different sets of data. JUnit 5 includes parameterized tests, which allow you to specify the source of that data, with options including comma-separated values (CSV) and factory methods. Even though JUnit is a Java library, tests can be written in Kotlin and used to test Kotlin code (as was done in much of this book).

Consider a function that computes Fibonacci numbers, as implemented using a tail-recursive algorithm in Example 9-13.

Tail recursion is discussed in Recipe 4.3.

Example 9-13. A tail-recursive function to compute the nth Fibonacci number

```
@JvmOverloads
tailrec fun fibonacci(n: Int, a: Int = 0, b: Int = 1): Int =
    when (n) {
        0 -> a
        1 -> b
        else -> fibonacci(n - 1, b, a + b)
    }
```

A Fibonacci number is defined as being the sum of the previous two Fibonacci numbers, where `fibonacci(0) == 0` and `fibonacci(1) == 1`.[1] The numbers form a series: 1, 1, 2, 3, 5, 8, 11, and so on.

Clearly, this has to be tested. An explicit test that simply calls the function multiple times is shown in Example 9-14.

Example 9-14. Explicitly calling the Fibonacci function

```
@Test
fun `Fibonacci numbers (explicit)`() {
    assertAll(
        { assertThat(fibonacci(4), `is`(3)) },
        { assertThat(fibonacci(9), `is`(34)) },
        { assertThat(fibonacci(2000), `is`(1392522469)) }
    )
}
```

JUnit 5 defines an `assertAll` function to ensure that all the included tests are executed, even if some fail, so this works. As an alternative, you can reformulate the test as a parameterized test by using a CSV source, as in Example 9-15.

1 Mandatory joke: This year's Fibonacci conference is going to be as good as the last two combined!

Example 9-15. Using CSV data to perform a parameterized test

```
@ParameterizedTest
@CsvSource("1, 1", "2, 1", "3, 2",
    "4, 3", "5, 5", "6, 8", "7, 13",
    "8, 21", "9, 34", "10, 55")
fun `first 10 Fibonacci numbers (csv)`(n: Int, fib: Int) =
     assertThat(fibonacci(n), `is`(fib))
```

The `@CsvSource` annotation takes as its argument a list of strings that are the input data for the function. Each string provides all the needed arguments to the function, which are separated by commas. This example checks the first 10 Fibonacci numbers, and the result is as follows:

```
 [1] 1, 1      first 10 Fibonacci numbers (csv)(int, int)[1]   0s      passed
 [2] 2, 1      first 10 Fibonacci numbers (csv)(int, int)[2]   0s      passed
 [3] 3, 2      first 10 Fibonacci numbers (csv)(int, int)[3]   0s      passed
 [4] 4, 3      first 10 Fibonacci numbers (csv)(int, int)[4]   0s      passed
 [5] 5, 5      first 10 Fibonacci numbers (csv)(int, int)[5]   0s      passed
 [6] 6, 8      first 10 Fibonacci numbers (csv)(int, int)[6]   0s      passed
 [7] 7, 13     first 10 Fibonacci numbers (csv)(int, int)[7]   0s      passed
 [8] 8, 21     first 10 Fibonacci numbers (csv)(int, int)[8]   0s      passed
 [9] 9, 34     first 10 Fibonacci numbers (csv)(int, int)[9]   0s      passed
[10] 10, 55    first 10 Fibonacci numbers (csv)(int, int)[10]  0s      passed
```

JUnit 5 also can use factory methods to generate the test data. In Java, a factory method in the test class must be `static` unless the test is annotated with `@TestInstance(Lifecycle.PER_CLASS)`, and if it is defined in an external class, it must always be `static`. It also cannot accept any arguments. Finally, the return type must be something the library knows how to iterate over, which includes streams, collections, iterables, iterators, or arrays.

If the life cycle is `Lifecycle.PER_CLASS`, as earlier recipes in this chapter use, then you can simply add a function to produce the data and reference it with `@MethodSource`, as in Example 9-16.

Example 9-16. Accessing an instance function as a parameter source

```
private fun fibnumbers() = listOf(
    Arguments.of(1, 1), Arguments.of(2, 1),
    Arguments.of(3, 2), Arguments.of(4, 3),
    Arguments.of(5, 5), Arguments.of(6, 8),
    Arguments.of(7, 13), Arguments.of(8, 21),
    Arguments.of(9, 34), Arguments.of(10, 55))

@ParameterizedTest(name = "fibonacci({0}) == {1}")
@MethodSource("fibnumbers")
fun `first 10 Fibonacci numbers (instance method)`(n: Int, fib: Int) =
    assertThat(fibonacci(n), `is`(fib))
```

JUnit provides the `Arguments` class with a factory method called `of` to put both input arguments together. The return type is `List<Arguments>`, where each element holds the two input arguments for the test method.

If the life cycle is the default `Lifecycle.PER_METHOD` you need to put the source function inside a companion object, as in Example 9-17.

Example 9-17. Using a companion object to hold the parameter source function

```
companion object {
// needed if parameterized test done with Lifecycle.PER_METHOD
    @JvmStatic                                                  ❶
    fun fibs() = listOf(
        Arguments.of(1, 1), Arguments.of(2, 1),
        Arguments.of(3, 2), Arguments.of(4, 3),
        Arguments.of(5, 5), Arguments.of(6, 8),
        Arguments.of(7, 13), Arguments.of(8, 21),
        Arguments.of(9, 34), Arguments.of(10, 55))
}

@ParameterizedTest(name = "fibonacci({0}) == {1}")
@MethodSource("fibs")
fun `first 10 Fibonacci numbers (companion method)`(n: Int, fib: Int) =
    assertThat(fibonacci(n), `is`(fib))
```

❶ Needed so the JUnit library (Java) will see `fun` as static

The only quirk here is that you need to add the `@JvmStatic` annotation so that the Java library JUnit will see the method source as a static method.

Finally, note that the `@ParameterizedTest` annotation takes a string argument that allows you to format the tests on the test report. The result for either set resembles this:

```
    fibonacci(1) == 1       first 10 Fibonacci numbers (method)(int, int)[1]
    fibonacci(2) == 1       first 10 Fibonacci numbers (method)(int, int)[2]
    fibonacci(3) == 2       first 10 Fibonacci numbers (method)(int, int)[3]
    fibonacci(4) == 3       first 10 Fibonacci numbers (method)(int, int)[4]
    fibonacci(5) == 5       first 10 Fibonacci numbers (method)(int, int)[5]
    fibonacci(6) == 8       first 10 Fibonacci numbers (method)(int, int)[6]
    fibonacci(7) == 13      first 10 Fibonacci numbers (method)(int, int)[7]
    fibonacci(8) == 21      first 10 Fibonacci numbers (method)(int, int)[8]
    fibonacci(9) == 34      first 10 Fibonacci numbers (method)(int, int)[9]
    fibonacci(10) == 55     first 10 Fibonacci numbers (method)(int, int)[10]
```

See Also

For more complex data, you can create a data class to hold it, which is discussed in Recipe 9.5.

9.5 Using Data Classes for Parameterized Tests

Problem

You want to produce easily readable test output for parameterized tests.

Solution

Create a data class that wraps the inputs and expected values, and use a function as a method source to generate the test data.

Discussion

JUnit 5 includes a capability known as *parameterized tests*: the test data is supplied by a method or a file, and each sample is run through the same test. Recipe 9.4 discusses that facility in detail. Each test in those examples, however, came down to an assertion that a processed value was equal to an expected one.

Consider the `fibonacci` function defined in Example 9-13, repeated here for simplicity:

```
    @JvmOverloads
    tailrec fun fibonacci(n: Int, a: Int = 0, b: Int = 1): Int =
        when (n) {
            0 -> a
            1 -> b
            else -> fibonacci(n - 1, b, a + b)
        }
```

The additional parameters `a` and `b` in this calculation are there to implement tail recursion and are given the appropriate default values. That means invoking the function is normally done with a single integer, and an integer is returned. Therefore, define a data class to hold the input and expected output, as in Example 9-18.

Example 9-18. Data class to hold input and expected output

```
data class FibonacciTestData(val number: Int, val expected: Int)
```

Because Kotlin data classes have a `toString` override already, you can build a test method for parameterized tests that instantiates the data class for each pair of inputs and outputs. See Example 9-19.

Example 9-19. Parameterized test using the data class

```
@ParameterizedTest
@MethodSource("fibonacciTestData")
fun `check fibonacci using data class`(data: FibonacciTestData) {
    assertThat(fibonacci(data.number), `is`(data.expected))
}

private fun fibonacciTestData() = Stream.of(
    FibonacciTestData(number = 1, expected = 1),
    FibonacciTestData(number = 2, expected = 1),
    FibonacciTestData(number = 3, expected = 2),
    FibonacciTestData(number = 4, expected = 3),
    FibonacciTestData(number = 5, expected = 5),
    FibonacciTestData(number = 6, expected = 8),
    FibonacciTestData(number = 7, expected = 13)
)
```

For JUnit tests that use a method source, the function must be `static` (in the Java sense) unless the test life cycle has been set to `TestInstance.Lifecycle.PER_CLASS`. Otherwise, move the private function into a companion object and mark it with `@JvmStatic` (see Recipe 9.1 for details).

The output from the test is as follows:

```
check fibonacci using data class(FibonacciTestData)
    [1] FibonacciTestData(number=1, expected=1)
    [2] FibonacciTestData(number=2, expected=1)
    [3] FibonacciTestData(number=3, expected=2)
    [4] FibonacciTestData(number=4, expected=3)
    [5] FibonacciTestData(number=5, expected=5)
    [6] FibonacciTestData(number=6, expected=8)
    [7] FibonacciTestData(number=7, expected=13)
```

The data class `FibonacciTestData` automatically defines a `toString` method that makes the results easy to read.

See Also

Parameterized tests in JUnit 5 are the subject of Recipe 9.4.

CHAPTER 10

Input/Output

Kotlin makes doing input/output (I/O) operations easy, but the style is different from what a Java developer may expect. Resources in Kotlin are frequently closed by employing a `use` function that does so on the user's behalf. This chapter includes a couple of recipes that focus on that approach, specifically for files, but that can be generalized to other resources.

10.1 Managing Resources with use

Problem

You want to process a resource such as a file and be sure that it is closed when you are finished, but Kotlin does not support Java's try-with-resources construct.

Solution

Use the extension functions `use` or `useLines` on readers or input/output streams.

Discussion

Java 1.7 introduced the try-with-resources construct, which allows a developer to open a resource inside parentheses between the `try` keyword and its corresponding block; the JVM will automatically close the resource when the `try` block ends. The only requirement is that the resource come from a class that implements the `Closeable` interface. `File`, `Stream`, and many other classes implement `Closeable`, as the example in Example 10-1 shows.

Example 10-1. Using try-with-resources from Java

```
package io;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class TryWithResourcesDemo {
    public static void main(String[] args) throws IOException {
        String path = "src/main/resources/book_data.csv";

        File file = new File(path);
        String line = null;
        try (BufferedReader reader = new BufferedReader(new FileReader(file))) {  ❶
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        }

        try (Stream<String> lines = Files.lines(Paths.get(path))) {  ❷
            lines.forEach(System.out::println);
        }
    }
}
```

❶ `BufferedReader` implements `Closeable`

❷ `Stream` implements `Closeable`

Both the `BufferedReader` class and `Stream` interface implement `Closeable`, so each has a `close` method that is automatically invoked when the `try` block completes.

A few interesting features are worth noting:

- In Java 10 or above, the declaration of `BufferedReader` and `Stream` could be replaced by the reserved word `var`. In fact, this is one of the primary use cases for local variable type inference. It is not used here to avoid confusion with Kotlin's `var` keyword.
- In Java 9 or above, you no longer have to create the `Closeable` variable inside the parentheses. It can be supplied from outside. Again, there is no benefit to doing so here.
- Since the `main` method signature was modified to throw `IOException`, the result is one of those rare moments when you have a `try` block without either a `catch` or a `finally`.

This is all well and good, but unfortunately the try-with-resources construct is not supported by Kotlin. Instead, Kotlin adds the extension functions `use` to `Closeable` and `useLines` to `Reader` and `File`.

The signature of `useLines` is given by the following:

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

The optional first argument is a character set, which defaults to UTF-8. The second argument is a lambda that maps a `Sequence` of lines from the file into a generic argument, `T`. The implementation of the `useLines` function automatically closes the reader after the processing is complete.

As an example, on Unix systems ultimately based on BSD (which include macOS), there is a file containing all the words from Webster's Second International Dictionary, which is out of copyright. On a Mac, the file is located in the directory */usr/share/dict/words*, and contains 238,000 words, one per line. Example 10-2 returns the 10 longest words in the dictionary.

Example 10-2. Finding the 10 longest words in the dictionary

```
fun get10LongestWordsInDictionary() =
    File("/usr/share/dict/words").useLines { line ->
        line.filter { it.length > 20 }
            .sortedByDescending(String::length)
            .take(10)
            .toList()
    }
```

The function filters out any line shorter than 20 characters (and since each line has a single word, that's any word shorter than 20 characters), sorts them by length in descending order, selects the first 10, and returns them in a list.

To call this function, use the following:

```
get10LongestWordsInDictionary().forEach { word ->
    println("$word (${word.length})")
```

This prints the following:

```
formaldehydesulphoxylate (24)
pathologicopsychological (24)
scientificophilosophical (24)
tetraiodophenolphthalein (24)
thyroparathyroidectomize (24)
anthropomorphologically (23)
blepharosphincterectomy (23)
epididymodeferentectomy (23)
```

```
    formaldehydesulphoxylic (23)
    gastroenteroanastomosis (23)
```

The implementation of `File.useLines` in the standard library is given by Example 10-3.

Example 10-3. Implementation of `useLines` as an extension function on `File`

```
inline fun <T> Reader.useLines(
    block: (Sequence<String>) -> T): T =
    buffered().use { block(it.lineSequence()) }
```

Note that the implementation creates a buffered reader (returned by the `buffered` function) and delegates to its `use` function.

The corresponding signature of the `use` function is given by Example 10-4.

Example 10-4. Signature of the `use` extension function on `Closeable`

```
inline fun <T : Closeable?, R> T.use(block: (T) -> R): R
```

The implementation is complicated by the need for exception handling, but at its base it is similar to this:

```
    try {
        return block(this)
    } catch (e: Throwable) {
        // save the exception to be used later
        throw e
    } finally {
        close() // requires another try/catch block
    }
```

The `use` block is an example of the Execute Around Method design pattern, where the infrastructure code is built into the library, and a provided lambda does the actual work. This separation of infrastructure from business logic makes it easier to focus on the task at hand.

The `use` function in this recipe is defined on `Closeable`, which is available in Java 6. If you require the underlying JDK to support Java 8, the same function is defined in `AutoCloseable` as well.

See Also

Recipe 10.2 shows how to code with the `use` block directly. The `use` function is also employed to shut down a Java thread pool in Recipe 13.4.

10.2 Writing to a File

Problem

You want to write to a file.

Solution

In addition to the normal Java input/output (I/O) methods, extension functions to the `File` class return output streams and writers.

Discussion

Several extension functions have been added to Java's `java.io.File` class. You can iterate through a file by using the `forEachLine` function. You can call `readLines` on a file to get a collection containing all the lines in the file, which is useful if the file is not very large. The `useLines` function, described in Recipe 10.1, allows you to supply a function that will be invoked on each line. If the file is small enough, you can use `readText` or `readBytes` to read the entire contents into a string or a byte array, respectively.

If you want to write to a file and replace all of its existing contents, use the `writeText` function, as in Example 10-5.

Example 10-5. Replacing all the text in a file

```
File("myfile.txt").writeText("My data")
```

It's hard to be much simpler than that. The `writeText` function takes an optional parameter to represent the character set, whose default value is UTF-8.

The `File` class also has an extension function called `appendText`, which can add data to a given file.

The `writeText` and `appendText` functions delegate to `writeBytes` and `appendBytes`, each of which takes advantage of the `use` function to ensure that the file is closed when writing is finished.

You can also use the `writer` (or `printWriter`) and `bufferedWriter` functions, which return an `OutputStreamWriter` and a `BufferedWriter`, as you might expect. With either of them, you can add a `use` block to do the actual writing, as in Example 10-6.

Example 10-6. Writing with the `use` function

```
File(fileName).printWriter().use { writer ->
    writer.println(data) }
```

Using `bufferedWriter` with a `use` block works exactly the same way.

See Also

Recipe 10.1 goes into detail about the `use` and `useLines` functions.

CHAPTER 11
Miscellaneous

This chapter consists of recipes that don't fit any of the other headings. Here you'll find how to make the `when` function exhaustive, how to measure the elapsed time of a function, and how to use the `TODO` function from the standard library, among many others.

11.1 Working with the Kotlin Version

Problem

You want to find out programmatically which version of Kotlin you are currently using.

Solution

Use the `CURRENT` property in the companion object of the `KotlinVersion` class.

Discussion

Since version 1.1, the *kotlin* package includes a class called `KotlinVersion` that wraps the major, minor, and patch values for the version number. Its `toString` method returns the combination as *major.minor.patch* given an instance of this class. The current instance of the class is contained in a public field called `CURRENT` in the companion object.

It's therefore trivially easy to return the current Kotlin version. Just access the field `KotlinVersion.CURRENT`, as in Example 11-1.

Example 11-1. Printing the current Kotlin version

```
fun main(args: Array<String>) {
    println("The current Kotlin version is ${KotlinVersion.CURRENT}")
}
```

The result is the major/minor/patch version of the Kotlin compiler, such as `1.3.41`. All three parts of this quantity are integers between 0 and `MAX_COMPONENT_VALUE`, which has the value 255.

The `CURRENT` property is annotated as a public `@JvmField` in the source code, so it is available from Java as well.

The `KotlinVersion` class implements the `Comparable` interface. That means you can use operators like `<` or `>` with it. The class also implements both `equals` and `hashCode`. Finally, the constructors for `KotlinVersion` allow you to supply either a `major` and a `minor` value, or a `major`, a `minor`, and a `patch` value.

As a result, you can do any of the following shown in Example 11-2.

Example 11-2. Comparing Kotlin versions

```
@Test
fun `comparison of KotlinVersion instances work`() {
    val v12 = KotlinVersion(major = 1, minor = 2)
    val v1341 = KotlinVersion(1, 3, 41)
    assertAll(
        { assertTrue(v12 < KotlinVersion.CURRENT) },
        { assertTrue(v1341 <= KotlinVersion.CURRENT) },
        { assertEquals(KotlinVersion(1, 3, 41),
            KotlinVersion(major = 1, minor = 3, patch = 41)) }
    )
}
```

The `isAtLeast` function is also available, to check whether a particular version of Kotlin is not less than major, minor, and patch values, as shown in Example 11-3.

Example 11-3. Checking that a version is above given values

```
@Test
fun `current version is at least 1_3`() {
    assertTrue(KotlinVersion.CURRENT.isAtLeast(major = 1, minor = 3))
    assertTrue(KotlinVersion.CURRENT.isAtLeast(major = 1, minor = 3, patch = 40))
}
```

It is therefore easy enough to check the Kotlin version and work with it directly.

11.2 Executing a Lambda Repeatedly

Problem

You want to execute a given lambda expression multiple times.

Solution

Use the built-in `repeat` function.

Discussion

The `repeat` function is in the standard library. It is an inline function that takes two arguments: an `Int` representing the number of times to iterate, and a function of the form `(Int) -> Unit` to execute.

The current implementation looks like Example 11-4.

Example 11-4. Definition of the `repeat` function

```
@kotlin.internal.InlineOnly
public inline fun repeat(times: Int, action: (Int) -> Unit) {
    contract { callsInPlace(action) }

    for (index in 0 until times) {
        action(index)
    }
}
```

The function executes a provided lambda a specified number of times, providing a zero-based index of the current iteration as a parameter.

A trivial example is shown in Example 11-5.

Example 11-5. Using `repeat`

```
fun main(args: Array<String>) {
    repeat(5) {
        println("Counting: $it")
    }
}
```

The output is simply this:

```
Counting: 0
Counting: 1
Counting: 2
Counting: 3
Counting: 4
```

Using `repeat` rather than a loop is an illustration of an internal iterator, where the actual looping process is handled by the library.

11.3 Forcing when to Be Exhaustive

Problem

You want the compiler to force the `when` statement to have a clause for every possibility.

Solution

Add a simple extension property called `exhaustive` to a generic type that returns a value, and chain it to the `when` block.

Discussion

Like the `if` statement, a notable feature of the `when` clause is that it returns a value. It behaves similarly to Java's `switch` statement, but unlike Java, you don't need to break out of each section, nor do you need to declare a variable outside it if you want to return a value.

For example, say you want to print out the remainder when a number is divided by 3, as in Example 11-6.

Example 11-6. Remainder when a number is divided by 3

```
fun printMod3(n: Int) {
    when (n % 3) {
        0 -> println("$n % 3 == 0")
        1 -> println("$n % 3 == 1")
        2 -> println("$n % 3 == 2")
    }
}
```

If a `when` expression does not return a value, Kotlin does not require it to be exhaustive, and this is an example where that is useful. Mathematically, we know that the remainder can be only 0, 1, or 2, so this is in fact an exhaustive check, but the compiler can't make that leap. If this simple function is converted into an expression, as in Example 11-7, that becomes clear.

Example 11-7. Using `when` to return a value

```
fun printMod3SingleStatement(n: Int) = when (n % 3) {
    0 -> println("$n % 3 == 0")
    1 -> println("$n % 3 == 1")
    2 -> println("$n % 3 == 2")
    else -> println("Houston, we have a problem...") ❶
}
```

❶ Does not compile without `else` clause

The compiler requires the `else` clause in this expression, even though the `println` function does not return anything. The presence of the equals sign means there's an assignment, which means Kotlin requires an exhaustive conditional expression.

Since Kotlin interprets any return as forcing an `else` block, you can take advantage of that to force all `when` blocks to be exhaustive by making them automatically return a value. To do so, create an extension property called `exhaustive`, as shown in Example 11-8.

Example 11-8. Adding an `exhaustive` property to any object

```
val <T> T.exhaustive: T
    get() = this
```

This block adds the `exhaustive` property to any generic type `T`, with a custom getter method that returns the current object.

Now this property can be added to anything, including a `when` block, to force an artificial return. Example 11-9 shows how this is done.

Example 11-9. Remainder when a number is divided by 3 (exhaustive)

```
fun printMod3Exhaustive(n: Int) {
    when (n % 3) {
        0 -> println("$n % 3 == 0")
        1 -> println("$n % 3 == 1")
        2 -> println("$n % 3 == 2")
        else -> println("Houston, we have a problem...")
    }.exhaustive  ❶
}
```

❶ Property forces the compiler to require `else` clause

The `exhaustive` property at the end of the `when` block returns the current object, so the Kotlin compiler requires it to be exhaustive.

While the purpose of this example was to force `when` to be exhaustive, it is also a nice example of how adding a simple extension property to a generic type can be useful as well.

11.4 Using the replace Function with Regular Expressions

Problem

You want to replace all instances of a substring with a given value.

Solution

Use the `replace` function on `String`, which is overloaded to take either a `String` argument or a regular expression.

Discussion

The `String` class implements the `CharSequence` interface, which means there are actually two versions of the `replace` function defined for it, as shown in Example 11-10.

Example 11-10. Two overloads for `replace`

```
fun String.replace(
    oldValue: String,
    newValue: String,
    ignoreCase: Boolean = false
): String

fun CharSequence.replace(
    regex: Regex,
    replacement: String
): String
```

Each of these replaces all the occurrences of the matching string or regular expression with the supplied value. The `replace` function defined on `String` takes an optional argument about case sensitivity, which defaults to *not* ignoring case.

These two overloads can be confusing, because a user might assume that the first one (which takes a `String` argument) will treat the string as though it were a regular expression, but that turns out not to be the case. The test in Example 11-11 shows the differences between the two functions.

Example 11-11. Using `replace` with the two overloads

```
@Test
fun `demonstrate replace with a string vs regex`() {
    assertAll(
        { assertEquals("one*two*", "one.two.".replace(".", "*")) },
        { assertEquals("********", "one.two.".replace(".".toRegex(), "*")) }
    )
}
```

The first example replaces the (literal) dots with an asterisk, while the second example treats the dots as they would be handled by regular expressions, meaning any single character. The first option therefore replaces only the two dots with asterisks, while the second one replaces all the individual characters with asterisks.

In fact, two potential traps exist for Java developers here:

- The `replace` function replaces all occurrences, not just the first one. In Java, the equivalent method is called `replaceAll`.
- The overload with a string as the first argument does not interpret that string as a regular expression. Again, this is unlike the Java method behavior. If you meant for your string to be interpreted as a regular expression, first convert it by using the `toRegex` function.

To give a more interesting example, consider checking whether a string is a palindrome. *Palindromes* are defined as strings that are the same forward and backward, ignoring both case and punctuation. Note that you can implement the function in a Java style (i.e., "speaking Kotlin with a Java accent"), as in Example 11-12.

Example 11-12. Palindrome checker, written in Java style

```
fun isPal(string: String): Boolean {
    val testString = string.toLowerCase().replace("""[\W+]""".toRegex(), "")
    return testString == testString.reversed()
}
```

There is nothing wrong with this approach, and it works just fine. It changes the string to lowercase and then uses the regular expression version of `replace` to replace all "nonword characters" with empty strings. In a regular expression, `\w` would represent any word character, meaning lowercase `a-z`, uppercase `A-Z`, the numbers `0-9`, and underscores. The capitalized version of `\w` is `\W`, which is the opposite of `\w`.

An arguably more idiomatic version of this same function is shown in Example 11-13.

Example 11-13. Palindrome checker, Kotlin style

```
fun String.isPalindrome() =
    this.toLowerCase().replace("""[\W+]""".toRegex(), "")
        .let { it == it.reversed() }
```

The differences are as follows:

- In this case, `isPalindrome` is added as an extension function to `String`, so no argument is needed. Inside the implementation, the current string is referenced as `this`.
- The `let` function allows you to write the entire test as a single expression on the generated test string. The local variable `testString` is no longer needed.
- Because now the body is a single expression, the braces have been replaced with an equals sign, as often happens in Kotlin functions.

The two approaches work the same way, but you're more likely to encounter the second approach from more experienced Kotlin developers. It's no doubt best to be familiar with both techniques. Either way, the implementation uses the overload of `replace` that takes a regular expression as its first argument.

See Also

The `let` function is discussed in Recipes 7.3 and 7.4.

11.5 Converting to Binary String and Back

Problem

You want to convert a number to a binary string (or some other base), or parse such a string to an integer.

Solution

Use the `toString` or `toInt` function overloads that take a `radix` as an argument.

Discussion

The `StringsKt` class contains an inline extension function on `Int` called `toString` that takes a radix. Likewise, the same class contains an extension function on `Int` to go the other way. That means you can convert from an `Int` into a binary string (i.e., a string composed of 1s and 0s) by invoking that method, as in Example 11-14.

Example 11-14. Converting an `Int` to a binary string

```
@Test
internal fun toBinaryStringAndBack() {
    val str = 42.toString(radix = 2)
    assertThat(str, `is`("101010"))

    val num = "101010".toInt(radix = 2)
    assertThat(num, `is`(42))
}
```

The string produced by `toString(Int)` will truncate any leading zeros. If you don't want to do that, you can postprocess the string by using the `padStart` function.

Say that you need to encode data by a single binary property. For example, playing cards come in red and black colors, and you want all permutations of four consecutive cards. That's a simple matter of counting from 0 to 15 in binary (with 0 representing red and 1 for black, or the other way around), but you don't want to lose the leading zeros in that case. You can solve that problem as shown in Example 11-15.

Example 11-15. Padding binary strings

```
@Test
internal fun paddedBinaryString() {
    val strings = (0..15).map {
        it.toString(2).padStart(4, '0')
    }

    assertThat(strings, contains(
        "0000", "0001", "0010", "0011",
        "0100", "0101", "0110", "0111",
        "1000", "1001", "1010", "1011",
        "1100", "1101", "1110", "1111"))

    val nums = strings.map { it.toInt(2) }
    assertThat(nums, contains(
        0, 1, 2, 3,
        4, 5, 6, 7,
        8, 9, 10, 11,
        12, 13, 14, 15))
}
```

Because the `toString` and `toInt` functions work for all integer bases, you don't have to restrict yourself to binary, though that's probably the most common use case. That means a nice variation on the classic joke (mentioned in Example 2-27) is as follows:

```
    val joke = """
        There are ${3.toString(3)} kinds of developers:
            - Those who know binary,
            - Those who don't, and
            - Those who didn't realize this is actually a ternary joke"""

    println(joke)
```

That prints the following:

There are 10 kinds of developers:

- Those who know binary,
- Those who don't, and
- Those who didn't realize this is actually a ternary joke.

11.6 Making a Class Executable

Problem

You have a class that contains a single function, and you want to make invoking that function trivial.

Solution

Override the `invoke` operator function on the class to call the function.

Discussion

Many operators in Kotlin can be overridden. To do so, you only need to override the function associated with that operator.

The Kotlin reference docs refer to this as *operator overloading*, but the concept is the same.

To implement an operator, you provide a member function or an extension function with the proper name and arguments. Any function that overloads operators needs to include the `operator` modifier.

One particular function is special: `invoke`. The `invoke` operator function allows instances of a class to be called as functions.

As an example, consider the free RESTful web service provided by Open Notify that returns JSON data representing the number of astronauts in space at any given moment. An example of the returned data is shown in Example 11-16.

Example 11-16. JSON data returned by the Open Notify service

```
{
    "people": [
        { "name": "Oleg Kononenko", "craft": "ISS" },
        { "name": "David Saint-Jacques", "craft": "ISS" },
        { "name": "Anne McClain", "craft": "ISS" }
    ],
    "number": 3,
    "message": "success"
}
```

The response shows that three astronauts are currently aboard the International Space Station.

The nested JSON objects imply that to parse this structure, two Kotlin classes are required, as shown in Example 11-17.

Example 11-17. Data classes modeling the returned JSON data

```
data class AstroResult(
    val message: String,
    val number: Number,
    val people: List<Assignment>
)

data class Assignment(
    val craft: String,
    val name: String
)
```

The `Assignment` class is the combination of astronaut name and craft. The `AstroResult` class is used for the overall response, which includes (hopefully) the "success" message, the number of astronauts, and their assignments.

If all you need is a simple HTTP GET request, Kotlin added an extension function called `readText` to the `java.net.URL` class. Invoking the sample service is therefore as simple as calling

```
    var response = URL("http://...").readText()
```

and processing the resulting JSON string. Since the desired URL is a constant and you can use any library, such as Google's Gson, to parse the JSON data, a reasonable class for accessing the service is given in Example 11-18.

Example 11-18. Accessing the RESTful service and parsing the result

```
import com.google.gson.Gson
import java.net.URL
```

```
class AstroRequest {
    companion object {
        private const val ASTRO_URL =
            "http://api.open-notify.org/astros.json"
    }

    // fun execute(): AstroResult {          ❶
    operator fun invoke(): AstroResult {     ❷
        val responseString = URL(ASTRO_URL).readText()
        return Gson().fromJson(responseString,
            AstroResult::class.java)
    }
}
```

❶ Arbitrary name for included function

❷ Operator function `invoke` makes class executable

In this class, the URL for the service is added to the companion object and specified to be a constant. The single function is used to access the service and provide the resulting string to Gson for parsing into an instance of `AstroResult`.

The function could be called anything. If it had been called `execute`, for instance, then invoking it would be done as follows:

```
    val request = AstroRequest()
    val result = request.execute()
    println(result.message)
```

And so on. There's nothing wrong with that approach, but observe that the class exists only to contain the single function. Kotlin is fine with using top-level functions, but it seems appropriate to include the URL as a constant as well. In other words, it is natural to create a class like `AstroRequest` as shown.

Since there is only one purpose for the class, changing the name of the function to `invoke` and adding the keyword `operator` to it makes the class itself executable, as Example 11-19 shows.

Example 11-19. Using the executable class

```
internal class AstroRequestTest {
    val request = AstroRequest()      ❶

    @Test
    internal fun `get people in space`() {
        val result = request()        ❷
        assertThat(result.message, `is`("success"))
        assertThat(result.number.toInt(),
            `is`(greaterThanOrEqualTo(0)))
        assertThat(result.people.size,
            `is`(result.number.toInt()))
    }
}
```

❶ Instantiates the class

❷ Invokes the class as a function (calls `invoke`)

Because `AstroResult` and `Assignment` are data classes, you can always just print the result, which looks like this:

```
AstroResult(message=success, number=3,
    people=[Assignment(craft=ISS, name=Oleg Kononenko),
    Assignment(craft=ISS, name=David Saint-Jacques),
    Assignment(craft=ISS, name=Anne McClain)])
```

The tests verify the individual properties.

By supplying the `invoke` operator function, the instance can be executed directly by adding parentheses to a reference. If desired, you can also add overloads of the `invoke` function with any needed arguments.

See also

Recipe 3.5 discusses operator overloading in more detail.

11.7 Measuring Elapsed Time

Problem

You want to know how long a code block takes to run.

Solution

Use either the `measureTimeMillis` or `measureNanoTime` functions in the standard library.

Discussion

The *kotlin.system* package includes the `measureTimeMillis` and `measureNanoTime` functions. Using them to determine how long a block takes to run is quite simple, as shown in Example 11-20.

Example 11-20. Measuring elapsed time for a code block

```
fun doubleIt(x: Int): Int {
    Thread.sleep(100L)
    println("doubling $x with on thread ${Thread.currentThread().name}")
    return x * 2
}

fun main() {
    println("${Runtime.getRuntime().availableProcessors()} processors")

    var time = measureTimeMillis {
        IntStream.rangeClosed(1, 6)
            .map { doubleIt(it) }
            .sum()
    }
    println("Sequential stream took ${time}ms")

    time = measureTimeMillis {
        IntStream.rangeClosed(1, 6)
            .parallel()
            .map { doubleIt(it) }
            .sum()
    }
    println("Parallel stream took ${time}ms")
}
```

The output of this snippet resembles the following:

```
    This machine has 8 processors
    doubling 1 with on thread main
    doubling 2 with on thread main
    doubling 3 with on thread main
    doubling 4 with on thread main
    doubling 5 with on thread main
    doubling 6 with on thread main
    Sequential stream took 616ms
    doubling 3 with on thread ForkJoinPool.commonPool-worker-11
    doubling 4 with on thread main
    doubling 5 with on thread ForkJoinPool.commonPool-worker-7
    doubling 6 with on thread ForkJoinPool.commonPool-worker-3
    doubling 2 with on thread ForkJoinPool.commonPool-worker-5
    doubling 1 with on thread ForkJoinPool.commonPool-worker-9
    Parallel stream took 110ms
```

Since the JVM reports eight processors, the `parallel` function on a stream splits the work among them and each processor gets a single element to double. Thus the result is that running the operation in parallel takes only about 100 milliseconds, while running it sequentially takes about 600 milliseconds.

The implementation of the `measureTimeMillis` function in the standard library is shown in Example 11-21.

Example 11-21. Implementation of the `measureTimeMillis` function

```
public inline fun measureTimeMillis(block: () -> Unit): Long {
    val start = System.currentTimeMillis()
    block()
    return System.currentTimeMillis() - start
}
```

Because it takes a lambda as an argument, this is a higher-order function, so as is typical, it is inlined for efficiency. The implementation just delegates to Java's `System.currentTimeMillis` method before and after executing the block argument. The implementation of `measureNanoTime` does the same thing, but delegates to `System.nanoTime`.

These two functions make it easy to do a simple profile of code performance. For a better estimate, consider the Java Microbenchmark Harness (JMH) project at OpenJDK.

11.8 Starting Threads

Problem

You want to run code blocks on concurrent threads.

Solution

Use the `thread` function in the *kotlin.concurrent* package.

Discussion

Kotlin provides a trivial extension function called `thread` that can be used to create and start threads easily. The signature of `thread` is shown here:

```
fun thread(
    start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit
): Thread
```

Because `start` defaults to `true`, this makes it easy to create and start multiple threads, as in Example 11-22.

Example 11-22. Starting multiple threads at random intervals

```
(0..5).forEach { n ->
    val sleepTime = Random.nextLong(range = 0..1000L)
    thread {
        Thread.sleep(sleepTime)
        println("${Thread.currentThread().name} for $n after ${sleepTime}ms")
    }
}
```

This code starts six threads, each of which sleeps for a random number of milliseconds between 0 and 1,000, and then prints the name of the thread. The output resembles this:

```
    Thread-2 for 2 after 184ms
    Thread-5 for 5 after 207ms
    Thread-4 for 4 after 847ms
    Thread-0 for 0 after 917ms
    Thread-3 for 3 after 967ms
    Thread-1 for 1 after 980ms
```

Note you don't need to call `start` to start each thread, since the `start` parameter in the `thread` function is true by default.

The `isDaemon` parameter lets you create daemon threads. If all the remaining threads in an application are daemon threads, the application can shut down. In other words, if the code in the previous example is replaced by that in Example 11-23, there will be no output at all, because the `main` function will exit before any threads complete.

Example 11-23. Starting daemon threads

```
(0..5).forEach { n ->
    val sleepTime = Random.nextLong(range = 0..1000L)
    thread(isDaemon = true) {        ❶
        Thread.sleep(sleepTime)
        println("${Thread.currentThread().name} for $n after ${sleepTime}ms")
    }
}
```

❶ Threads are daemon threads, so program exits before threads finish running

The required code block is a lambda that takes no arguments and returns `Unit`. This is consistent with the `Runnable` interface, or simply the signature of the `run` method in `Thread`. The implementation of the `thread` function is shown in Example 11-24.

Example 11-24. Implementation of the `thread` function in the standard library

```
public fun thread(
    start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit
): Thread {
    val thread = object : Thread() {
        public override fun run() {
            block()
        }
    }
    if (isDaemon)
        thread.isDaemon = true
    if (priority > 0)
        thread.priority = priority
    if (name != null)
        thread.name = name
    if (contextClassLoader != null)
        thread.contextClassLoader = contextClassLoader
    if (start)
        thread.start()
    return thread
}
```

The implementation creates an object of type `Thread` and overrides its `run` method to invoke the supplied `block`. It then sets the various supplied properties and calls `start`.

Because the function returns the created thread, you can make all the threads run sequentially by invoking the `join` method on them, as in Example 11-25.

Example 11-25. Joining the threads together

```
(0..5).forEach { n ->
    val sleepTime = Random.nextLong(range = 0..1000L)
    thread {
        Thread.sleep(sleepTime)
        println("${Thread.currentThread().name} for $n after ${sleepTime}ms")
    }.join()  ❶
}
```

❶ Causes each thread to join the previous one

The output will now resemble the following:

```
    Thread-0 for 0 after 687ms
    Thread-1 for 1 after 661ms
    Thread-2 for 2 after 430ms
    Thread-3 for 3 after 412ms
    Thread-4 for 4 after 918ms
    Thread-5 for 5 after 755ms
```

Of course, that obviates the need to run inside threads in the first place, but it demonstrates that you can invoke methods on the returned threads.

See Also

Chapter 13 discusses concurrency in much more detail.

11.9 Forcing Completion with TODO

Problem

You want to guarantee that you complete a particular function or test.

Solution

Use the `TODO` function (with an optional reason) that throws an exception if you don't complete a function.

Discussion

Developers often leave notes to themselves to complete a function that they're not ready to finish at the moment. In most languages, you add a "TODO" statement in a comment, as in this example:

```
fun myCleverFunction() {
    // TODO: look up cool implementation
}
```

The Kotlin standard library includes a function called `TODO`, the implementation of which is shown in Example 11-26.

Example 11-26. Implementation of the `TODO` function

```
public inline fun TODO(reason: String): Nothing =
    throw NotImplementedError("An operation is not implemented: $reason")
```

The source is inlined for efficiency and throws a `NotImplementedError` when invoked. In regular source code, it's easy enough to use, as in Example 11-27.

Example 11-27. Using the `TODO` function in regular code

```
fun main() {
    TODO(reason = "none, really")
}

fun completeThis() {
    TODO()
}
```

The result of executing this script is as follows:

```
    Exception in thread "main" kotlin.NotImplementedError:
        An operation is not implemented: none, really
            at misc.TodosKt.main(todos.kt:4)
            at misc.TodosKt.main(todos.kt)
```

The optional `reason` argument can explain what the developer intends.

The `TODO` function can also be used in a test, with an expected exception until the test is completed, as in Example 11-28.

Example 11-28. Using `TODO` in a test

```
fun `todo test`() {
    val exception = assertThrows<NotImplementedError> {
        TODO("seriously, finish this")
    }
    assertEquals("An operation is not implemented: seriously, finish this",
        exception.message)
}
```

The `TODO` function is one of those convenient additions to the library that are easy to overlook until someone points it out to you. Hopefully, you can find ways to take advantage of it.

11.10 Understanding the Random Behavior of Random

Problem

You want to generate a random number.

Solution

Use one of the functions in the `Random` class.

Discussion

The basics of `kotlin.random.Random` are straightforward, but the implementation is quite subtle. First, the easy part. If you want a random `Int`, use one of the overloads of `nextInt`. The documentation for `kotlin.random.Random` states that it is an abstract class, but includes the methods in Example 11-29.

Example 11-29. Declarations in the abstract `Random` class

```
open fun nextInt(): Int
open fun nextInt(until: Int): Int
open fun nextInt(from: Int, until: Int): Int
```

All three of these functions are given a default implementation. Using them is easy enough, as shown in Example 11-30.

Example 11-30. The overloads of the `nextInt` function

```
@Test
fun `nextInt with no args gives any Int`() {
    val value = Random.nextInt()
    assertTrue(value in Int.MIN_VALUE..Int.MAX_VALUE)
}

@Test
fun `nextInt with a range gives value between 0 and limit`() {
    val value = Random.nextInt(10)
    assertTrue(value in 0..10)
}

@Test
fun `nextInt with min and max gives value between them`() {
    val value = Random.nextInt(5, 10)
    assertTrue(value in 5..10)
}

@Test
fun `nextInt with range returns value in range`() {
    val value = Random.nextInt(7..12)
    assertTrue(value in 7..12)
}
```

That last example, however, is not listed as one of the functions in the `Random` class. Instead, it's an extension function, whose signature is shown here:

```
    fun Random.nextInt(range: IntRange): Int
```

The result is that if you look at the import statements in the previous test cases, you'll see that they import both `kotlin.random.Random` and `kotlin.random.nextInt`, the latter being the extension function.

The implementation of the `Random` class is quite interesting. The methods listed in Example 11-29 are provided, followed by a companion object of type `Random`. A snippet from the implementation is shown in Example 11-31.

Example 11-31. Companion object inside `Random`

```
companion object Default : Random() {
    private val defaultRandom: Random = defaultPlatformRandom()

    override fun nextInt(): Int = defaultRandom.nextInt()
    override fun nextInt(until: Int): Int = defaultRandom.nextInt(until)
    override fun nextInt(from: Int, until: Int): Int =
        defaultRandom.nextInt(from, until)

// ...
}
```

The companion object gets the default implementation, and overrides all the declared methods to delegate to the default. The implementation of the `defaultPlatformRandom` function is internal.

The same pattern is included for other types, like `Boolean`, `Byte`, `Float`, `Long`, and `Double`, as well as the unsigned types `UBytes`, `UInt`, and `ULong`.

To make things even more fun, there is also a function called `Random` that takes an `Int` or `Long` seed, which returns a repeatable random number generator seeded with the argument. See Example 11-32.

Example 11-32. Using a seeded random number generator

```
@Test
fun `Random function produces a seeded generator`() {
    val r1 = Random(12345)
    val nums1 = (1..10).map { r1.nextInt() }

    val r2 = Random(12345)
    val nums2 = (1..10).map { r2.nextInt() }

    assertEquals(nums1, nums2)
}
```

Given the same seed, the calls to `nextint` provide the same sequence of random numbers.

Using the random number generators in Kotlin is straightforward, but examining the implementation of the methods (using an abstract class whose methods are overridden inside its own companion object) may give you ideas about how to design your own classes in the future.

11.11 Using Special Characters in Function Names

Problem

You want to write function names that are easy to read.

Solution

You can use underscores or surround your function name in backticks, but only in tests.

Discussion

Kotlin supports wrapping the names of functions inside backticks, as shown in Example 11-33.

Example 11-33. Wrapping function names inside backticks

```kotlin
fun `only use backticks on test functions`() {
    println("This works but is not a good idea")
}

fun main() {
    `only use backticks on test functions`()
}
```

Wrapping the function name inside backticks allows you to put spaces in the name for readability. If you do this in a regular function, IntelliJ IDEA will flag it, saying, "Function name may only contain letters and digits." So the function will compile and run, as shown, but isn't a good practice.

Another option is to use underscores, as in Example 11-34.

Example 11-34. Using underscores in function names

```kotlin
fun underscores_are_also_okay_only_on_tests() {
    println("Again, please don't do this outside of tests")
}

fun main() {
    underscores_are_also_okay_only_on_tests()
}
```

Again, this will compile and run, but the compiler will issue a warning like, "Function name should not contain underscores."

On the other hand, you can use either mechanism inside tests, and it counts as idiomatic Kotlin (as shown in the Coding Conventions guide). So the example shown in Example 11-35 is fine.

Example 11-35. Test function names can use either technique

```
class FunctionNamesTest {
    @Test
    fun `backticks make for readable test names`() {
        // ...
    }

    @Test
    fun underscores_are_fine_here_too() {
        // ...
    }
}
```

Even better, the provided readable names will show up on the test report, and anything that makes testing clearer and easier is a good thing.

11.12 Telling Java About Exceptions

Problem

Your Kotlin function throws what Java would consider a checked exception, but you need to tell Java that.

Solution

Add a `@Throws` annotation to the function signature.

Discussion

All exceptions in Kotlin are considered unchecked, meaning the compiler does not require you to handle them. It's easy enough to add try/catch/finally blocks to a Kotlin function if you wish to catch an exception, but you are not forced to do so.

> Kotlin doesn't have the `throws` keyword that Java uses to declare that a method may throw an exception.

That's fine until you try to invoke that function from Java. If the Kotlin function potentially throws an exception that Java would consider checked, you need to let Java know about it if you want to catch it.

For example, say you have a Kotlin function that throws an `IOException`, which is a checked exception in Java, shown in Example 11-36.

Example 11-36. A Kotlin function that throws an `IOException`

```
fun houstonWeHaveAProblem() {
    throw IOException("File or resource not found")
}
```

In Kotlin, this function does not need a try/catch block or a `throws` clause in order to compile. The function throws an `IOException`, as shown.

This function can be called from Java, and an exception will result, as in Example 11-37.

Example 11-37. Calling the Kotlin function from Java

```
public static void doNothing() {
    houstonWeHaveAProblem();  ❶
}
```

❶ Crashes with an `IOException`

(The source code for this example includes a static import for the invoked function.)

The problem comes if you decide you want to prepare for the `IOException` by either wrapping the call inside a try/catch block or adding a `throws` clause to the Java call, as in Example 11-38.

Example 11-38. Trying to prepare for the expected exception

```
public static void useTryCatchBlock() {
    try {                           ❶
        houstonWeHaveAProblem();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void useThrowsClause() throws IOException { ❷
    houstonWeHaveAProblem();
}
```

❶ Does not compile

❷ Compiles, but compiler warns about "unnecessary" `throws` clause

Neither of these work the way you want. If you try to add an explicit try/catch block, the code won't compile because Java thinks the specified `IOException` in the `catch` block is never thrown in the corresponding `try` block. In the second case, the code will compile, but your IDE (and the compiler) will warn you that you have "unnecessary" code.

The way to make either approach work is to add a `@Throws` annotation to the Kotlin code, as in Example 11-39.

Example 11-39. Adding a `@Throws` annotation

```
@Throws(IOException::class)    ❶
fun houstonWeHaveAProblem() {
    throw IOException("File or resource not found")
}
```

❶ Tells Java this function throws an `IOException`

Now the Java compiler knows you need to prepare for the `IOException`. Of course, the `doNothing` function no longer compiles, because `IOException` is checked so you need to prepare for it.

The `@Throws` annotation exists simply for Java/Kotlin integration. It solves a specific problem, but it works exactly as advertised.

CHAPTER 12

The Spring Framework

The Spring framework is one of the most popular open source frameworks in the Java world. Spring is all about your project's infrastructure. You focus on writing *beans* that contain the business logic you need for your goals, and Spring provides services like security, transactions, resource pooling, and more, based on metadata you provide.

Spring has always been friendly with "alternative" languages on the JVM. Spring has had support for Groovy since at least version 2.5. In the past few versions, the developers of Spring have added capabilities unique to Kotlin as well.

This chapter contains a few select techniques you can use in Spring applications when writing your code in Kotlin. Spring support for Kotlin is still evolving, but this chapter should give you a sense of how Kotlin figures into the Spring ecosystem.

12.1 Opening Spring-Managed Bean Classes for Extension

Problem

Spring needs to generate proxies by extending your classes, but Kotlin classes are closed (`final` in Java) by default.

Solution

Add the Spring plug-in for Kotlin to your build file, which automatically opens the needed Spring-managed classes for extension.

Discussion

Spring works by providing services to your system. It does so through the Proxy design pattern. The Proxy design pattern, illustrated in UML class and sequence diagrams, is shown in Figure 12-1.

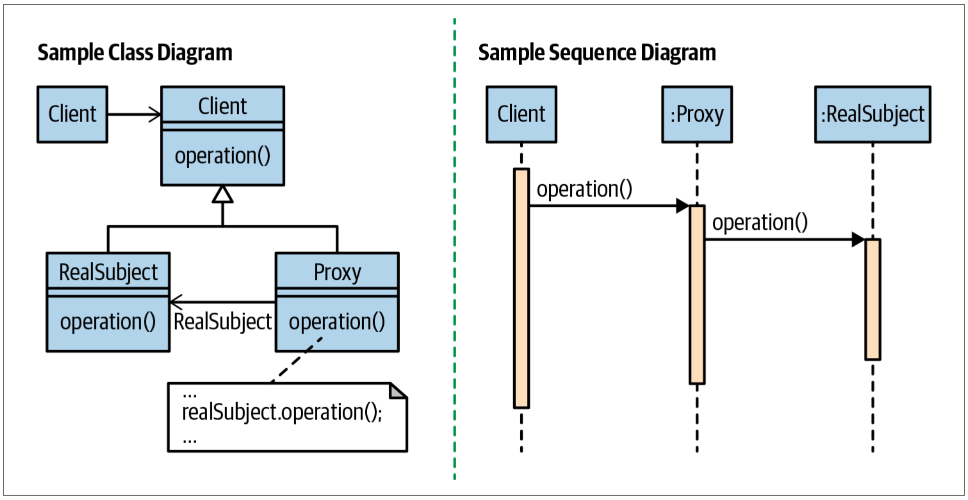


Figure 12-1. UML diagram of the Proxy design pattern

The idea is that the proxy and the real subject both either implement the same interface or extend the same class. An incoming request is intercepted by the proxy, which applies whatever services are desired, and then forwards the request to the real subject. The proxy can also intercept the response and do more work if necessary. For example, a Spring transactional proxy intercepts a call to a method, starts a transaction, invokes the method, and calls either `commit` or `rollback` depending on what happened inside the real subject's method.

Spring generates proxies during its start-up process. If the real subject is a class, it needs to extend that class, and that's where Kotlin has a problem. Kotlin is statically bound by default, meaning you cannot override a method, or even extend a class, unless it has been marked open for extension using the `open` keyword.

To handle problems like this, Kotlin has a plug-in called the *all-open plug-in*. This plug-in makes classes marked with a specified annotation open, without explicitly adding the `open` keyword to the class and the functions it contains.

While this is useful, the developers of the language have gone beyond this to create a *kotlin-spring* plug-in, which is naturally suited to Spring. To use it, add the plug-in to either your Gradle or Maven build file. The build snippet in Example 12-1 shows a Gradle build file (that uses the Kotlin DSL) that includes the plug-in. The file is called *build.gradle.kts*, and is generated by the Spring Initializr.

Example 12-1. Adding the kotlin-spring plug-in

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    id("org.springframework.boot") version "2.1.8.RELEASE"
    id("io.spring.dependency-management") version "1.0.8.RELEASE"
    kotlin("jvm") version "1.2.71"                                ❶
    kotlin("plugin.spring") version "1.2.71"                      ❷
}

group = "com.mycompany"
version = "1.0"

java.sourceCompatibility = JavaVersion.VERSION_11

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter")
    implementation("org.jetbrains.kotlin:kotlin-reflect")           ❸
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")  ❸
    testImplementation("org.springframework.boot:spring-boot-starter-test")
}

tasks.withType<KotlinCompile> {
    kotlinOptions {
      freeCompilerArgs = listOf("-Xjsr305=strict")               ❹
      jvmTarget = "1.8"
    }
}
```

❶ Add the Kotlin JVM plug-in to the project

❷ Requires the Kotlin Spring plug-in

❸ Required if the project source code is written in Kotlin

❹ Supports nullability annotations associated with JSR-305

The *all-open* plug-in lets you declare which annotations are used to open Kotlin classes. The *kotlin-spring* plug-in is already configured to work with the following Spring annotations:

- `@Component`
- `@Async`
- `@Transactional`

- `@Cacheable`
- `@SpringBootTest`

The `@Component` annotation is used in several other Spring-composed annotations, including `@Configuration`, `@Controller` and `@RestController`, `@Service`, and `@Repository`. All managed Spring beans marked with any of those annotations are automatically open for extension, which is normally all you need.

If you need to go beyond that, you still add the *all-open* plug-in as well, but that's typically not necessary.

To see the Maven build that employs the same plug-in, generate a Maven project by using the Initializr. The concepts are the same, however.

See Also

The *kotlin-jpa* plugin is discussed in Recipe 12.2.

12.2 Persisting Kotlin Data Classes

Problem

You want to use the Java Persistence API (JPA) with Kotlin data classes.

Solution

Add the *kotlin-jpa* plug-in to your build file.

Discussion

Generally, when you define a Kotlin data class, you add any necessary properties to the primary constructor, as in Example 12-2.

Example 12-2. A data class with a primary constructor

```kotlin
data class Person(val name: String,
    val dob: LocalDate)
```

There are two problems with this from a JPA perspective. First, JPA requires a default constructor, and unless you give default values to all your properties, this class doesn't have one. Second, by making it a data class with `val` properties, this produces immutable objects, and JPA isn't designed to play nicely with immutable objects.

Treating the default constructor problem first, Kotlin provides two plug-ins to address that issue. The *no-arg* plug-in lets you choose which classes should be given a no argument constructor and lets you define annotations to invoke them. Second, building on top of this, you have the *kotlin-jpa* plug-in. That plug-in automatically configures Kotlin *entities* (e.g., classes annotated with `@Entity`, among others in this recipe) with default constructors.

As with the *kotlin-spring* plug-in discussed in Recipe 12.1, you take advantage of these plug-ins by adding the required syntax to your build file. The recipe on the *kotlin-spring* plug-in showed a Gradle build file (using the Kotlin DSL). Building on that, make the following additions to *build.gradle.kts* from Example 12-1, shown in Example 12-3.

Example 12-3. Additional dependencies added for JPA entities

```
plugins {
    // ... as before ...
        kotlin("plugin.jpa") version "1.2.71"
}

// ... other settings as before ...

dependencies {
    // ... other dependencies from previous recipe ...
        implementation("org.springframework.boot:spring-boot-starter-data-jpa")
        implementation("com.fasterxml.jackson.module:jackson-module-kotlin")
}
```

The `jackson-module-kotlin` dependency shown is not necessary for entities, but it helps serialize Kotlin classes into JSON form and back by using the Jackson 2 library.

The *no-arg* compiler plug-in adds a *synthetic* default constructor to Kotlin classes, meaning it cannot be invoked in Java or Kotlin, but Spring can call it using reflection. You can use this plug-in if you want, but then you have to define which annotations will be used to mark classes that need the *no-arg* constructor.

It's easier to use the *kotlin-jpa* plug-in, as shown previously in Example 12-3. The *kotlin-jpa* plug-in is built on top of the *no-arg* plug-in. It automatically adds default constructors to any class annotated with the following:

- `@Entity`
- `@Embeddable`

- `@MappedSuperclass`

The other problem is that JPA doesn't want to work with immutable classes for entities. Therefore, the Spring team recommends that any Kotlin classes you want to use as entities should be simple classes (instead of data classes) with `var` types for the properties, so the field values can be changed. An example given in the Spring tutorial on using Spring Boot with Kotlin is shown in Example 12-4.

Example 12-4. Kotlin classes that map to database tables

```kotlin
@Entity
class Article(
    var title: String,
    var headline: String,
    var content: String,
    @ManyToOne var author: User,
    var slug: String = title.toSlug(),
    var addedAt: LocalDateTime = LocalDateTime.now(),
    @Id @GeneratedValue var id: Long? = null)

@Entity
class User(
    var login: String,
    var firstname: String,
    var lastname: String,
    var description: String? = null,
    @Id @GeneratedValue var id: Long? = null)
```

The `Article` and `User` classes shown use `var` on the properties, and even make the generated primary key field nullable. In Hibernate parlance (and the most common JPA provider is still Hibernate), a null primary key (annotated here with `@Id`) indicates that the instance is in the transient state, meaning there is no row in the corresponding database table associated with that instance. That happens either when you first instantiate the class and haven't yet saved it, or if you've deleted the row from the database and the instance is still in memory.

The `@GeneratedValue` annotation indicates that the database itself is providing values for the primary key.

As a Kotlin developer, the extensive use of `var` and the lack of automatically generated `toString`, `equals`, and `hashCode` functions feels uncomfortable. That said, this approach is more compatible with what JPA expects. If you use a different API based on Spring Data, like Spring Data MongoDB or Spring Data JDBC, you are free to use data classes instead.

See Also

The *kotlin-spring* plug-in is discussed in Recipe 12.1.

12.3 Injecting Dependencies

Problem

You want to autowire beans together using dependency injection, and declare which beans are required and which are not.

Solution

Kotlin provides constructor injection, but for field injection, use the `lateinit var` structure. Declare optional beans by using nullable types.

Discussion

You wire beans together in Spring via a process known as *dependency injection*, which sounds a lot more complicated than it is. The idea is that you add a reference of one type to a class of another type, and Spring will find a way to provide an instance of that type for you.

Spring favors constructor injection for dependencies wherever possible. In Kotlin, you can use the `@Autowired` annotation directly on constructor arguments. If you have only a single constructor in a class, you don't even need the `@Autowired` annotation, because all arguments to the single constructor will be autowired automatically.

If you have only a single constructor in a Spring-managed bean, Spring will inject all the arguments automatically.

Say you have a REST controller that works with an injected service. All the approaches to autowiring shown in Example 12-5 will work.

Example 12-5. Autowiring a dependency into Spring

```
@RestController    ❶
class GreetingController(val service: GreetingService) { /* ... */ }

@RestController    ❷
class GreetingController(@Autowired val service: GreetingService) { /* ... */ }

@RestController    ❸
```

```
class GreetingController @Autowired constructor(val service: GreetingService) {
    // ... (normal 4-space indent)
}

@RestController    ❹
class GreetingController {
    @Autowired
    lateinit var service: GreetingService

    // ... rest of class ...
}
```

❶ Option 1: Class with a single constructor

❷ Option 2: Explicit autowiring

❸ Option 3: Autowiring constructor call, primarily for classes with multiple dependencies

❹ Option 4: Field injection (not preferred, but can be useful)

The example shows all the ways to perform dependency injection:

- Simply declare the dependencies, and a class with only a single constructor will have its properties autowired automatically.
- Use the explicit `@Autowired` annotation, which works the same way, but the explicit statement of `@Autowired` will continue to work even if you add a secondary constructor.
- Put `@Autowired` on the `constructor` function, which is normally a simplification used when you have multiple dependencies to inject.
- Finally, if you must use field injection, use the `lateinit var` modifiers on the field.

Because `val` properties must have a value when they are declared, you can't initialize one with a value supplied later. That's why the `lateinit` keyword is used with `var`. The downside is that `var` properties, by definition, can be changed at any subsequent time, which may not be what you want. That's one reason, among others, why constructor injection is preferred.

If a property of a class is not required, you can declare it to be nullable. Kotlin will take that to mean it is optional. For example, consider a function in the `GreetingController` that generates a greeting from an optional request parameter, as shown in Example 12-6.

Example 12-6. Controller function with an optional parameter

```kotlin
@GetMapping("/hello")
fun greetUser(@RequestParam name: String?) =
    Greeting(service.sayHello(name?: "World")) else Greeting()
```

By declaring the parameter `name` to be nullable (e.g., of type `String?` rather than simply `String`), Kotlin knows that the parameter is not required.

Spring also advocates the use of JUnit 5 for tests. Two features provided by JUnit 5 that are not available in JUnit 4 are as follows:

- You can have a nondefault constructor in JUnit 5 tests.
- You can set the life cycle of the test class to be one instance per class, rather than reinstantiating it per method.

An example of this from the Kotlin/Spring tutorial shows a test that starts with the code in Example 12-7.

Example 12-7. Injecting dependencies by using constructor arguments in JUnit 5

```kotlin
@DataJpaTest
class RepositoriesTests @Autowired constructor(
    val entityManager: TestEntityManager,
    val userRepository: UserRepository,
    val articleRepository: ArticleRepository) {

    // ... tests go here ...
}

// Another test, using the Spring RestTemplate class
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class IntegrationTests(@Autowired val restTemplate: TestRestTemplate) {
    // ... tests go here ...
}
```

The use of the autowired constructor means you don't need to use the `lateinit var` approach. The first case, `RepositoriesTests`, autowires in classes created by the user. The second example, `IntegrationTests`, starts a test server on a random port and deploys the web application to it, then uses the `TestRestTemplate` class (which is already configured with the proper port) to make REST requests using functions like `getForObject` or `getForEntity`.

CHAPTER 13

Coroutines and Structured Concurrency

One of Kotlin's most popular features is its support for *coroutines*, which allow developers to write concurrent code as though it was synchronous. That support makes it much easier to write concurrent code that employs coroutines than using other techniques, like callback methods or reactive streams.

Note that the key word in that sentence is *easier*, rather than *easy*. Managing concurrency is always a challenge, especially when you try to coordinate multiple separate activities, handle cancellations and exceptions, and more.

This chapter discusses the issues related to Kotlin coroutines. These issues include working with coroutine scope and coroutine context, selecting the proper coroutine builder and dispatchers, and coordinating their behavior.

The idea behind coroutines is that they can be suspended and resumed. By marking a function with the `suspend` keyword, you're telling the system that it can put the function on hold temporarily, and resume it on another thread later, all without having to write complex multithreading code yourself.

13.1 Choosing Coroutine Builders

Problem

You need to select the right function to create a coroutine.

Solution

Decide between the available builder functions.

Discussion

To create a new coroutine, you use one of the available builder functions: `runBlocking`, `launch`, or `async`. The first, `runBlocking`, is a top-level function, while `launch` and `async` are extension functions on `CoroutineScope`.

Before looking at how they are used, be aware that there are also versions of `launch` and `async` defined on `GlobalScope`, and their usage is highly discouraged, if not completely deprecated. The problem with those functions is that they launch coroutines that are not bound to any particular job, and they span the entire application life cycle if not cancelled prematurely. So please don't use them unless you have an overriding reason to do so.

This section arguably could have been titled, "Choosing Coroutine Builders, and `GlobalScope.launch` Is the Wrong Answer."

The runBlocking builder

Returning to the recommended approaches, `runBlocking` is useful for command-line demonstrations or for tests. As the name indicates, it blocks the current thread and waits until all included coroutines have finished.

The signature of the `runBlocking` function is as follows:

```kotlin
fun <T> runBlocking(block: suspend CoroutineScope.() -> T): T
```

The `runBlocking` function is not itself a suspending function, so it can be called from normal functions. It takes a suspending function as an argument, which it adds as an extension function to `CoroutineScope`, executes it, and returns whatever value the supplied function returns.

Using `runBlocking` is quite simple, as Example 13-1 shows.

Example 13-1. Using the `runBlocking` function

```kotlin
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking

fun main() {
    println("Before creating coroutine")
    runBlocking {
        print("Hello, ")
        delay(200L)
        println("World!")
    }
    println("After coroutine is finished")
}
```

The output of this code is simply as follows:

```
    Before creating coroutine
    Hello, World!
    After coroutine finished
```

Note, however, that there is a 200-millisecond delay between printing “Hello,” and “World!”.

The launch builder

If you need to start a coroutine to execute a separate process but don’t need to return a value from it, use the `launch` coroutine builder. The `launch` function is an extension function on `CoroutineScope`, so it can be used only if a `CoroutineScope` is available. It returns an instance of `Job`, which can be used to cancel the coroutine if necessary.

The signature of the `launch` function is shown here:

```
    fun CoroutineScope.launch(
        context: CoroutineContext = EmptyCoroutineContext,
        start: CoroutineStart = CoroutineStart.DEFAULT,
        block: suspend CoroutineScope.() -> Unit
    ): Job
```

The `CoroutineContext` is used to share state with other coroutines. The `CoroutineStart` parameter is an enumerated class, whose values can be only `DEFAULT`, `LAZY`, `ATOMIC`, or `UNDISPATCHED`.

The supplied lambda must be a suspending function that takes no arguments and does not return anything. Example 13-2 shows the use of `launch`.

Example 13-2. Using the `launch` function

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
    println("Before runBlocking")
    runBlocking {                         ❶
        println("Before launch")
        launch {                          ❷
            print("Hello, ")
            delay(200L)
            println("World!")
        }
        println("After launch")
    }
    println("After runBlocking")
}
```

❶ Creates a coroutine scope

❷ Launches a coroutine

The output is what you would expect:

```
Before runBlocking
Before launch
After launch
Hello, World!
After runBlocking
```

Again, the string "Hello," is printed, and then, after a 200-millisecond delay, the string "World!".

Cancellation using the returned `Job` is discussed in Example 13-5.

The async builder

In the common situation where you need to return a value, use the `async` builder. It is also an extension function on `CoroutineScope`, and its signature is as follows:

```
fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

Again, the `CoroutineContext` and `CoroutineStart` parameters have reasonable defaults.

This time, the supplied suspending function does return a value, which the `async` function then wraps inside a `Deferred` instance. A `Deferred` instance feels like a promise in JavaScript, or a future in Java. The important function to know on `Deferred` is `await`, which waits until a coroutine has completed before returning the produced value.

A trivial example using `async` is shown in Example 13-3.

Example 13-3. Creating coroutines with async

```
import kotlinx.coroutines.async
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.delay
import kotlin.random.Random

suspend fun add(x: Int, y: Int): Int {
    delay(Random.nextLong(1000L))                               ❶
    return x + y
}

suspend fun main() = coroutineScope {                          ❷
    val firstSum = async {                                     ❸
        println(Thread.currentThread().name)
        add(2, 2)
    }
    val secondSum = async {                                    ❸
        println(Thread.currentThread().name)
        add(3, 4)
    }
    println("Awaiting concurrent sums...")
    val total = firstSum.await() + secondSum.await()           ❹
    println("Total is $total")
}
```

❶ Random delay up to 1,000 ms

❷ Another coroutine builder, discussed later in this recipe

❸ Uses `async` to launch a coroutine

❹ Invokes `await` to block until the coroutines finish

The `add` function delays executing for a random number of milliseconds less than 1,000, and then returns the sum. The two `async` calls invoke the `add` function and return instances of `Deferred`. The calls to `await` then block until the coroutines complete.

The result is shown here:

```
DefaultDispatcher-worker-2
Awaiting concurrent sums...
DefaultDispatcher-worker-1
Total is 11
```

Note that the `delay` function is a suspending function that puts a coroutine on hold without blocking the thread on which it is running.

The two `async` builders are using the default dispatcher, one of the dispatchers discussed in Recipe 13.3. The `runBlocking` call will wait until everything has completed before exiting the program. The order of the output lines depends on the randomly generated delays.

The coroutineScope builder

Finally, the `coroutineScope` function is a suspending function that waits until all included coroutines finish before exiting. It has the advantage of not blocking the main thread (unlike `runBlocking`), but it must be called as part of a `suspend` function.

This gets to one of the fundamental principles of using coroutines, which is to use them inside a defined scope. The benefit of `coroutineScope` is that you don't have to poll to see whether coroutines are finished—it automatically waits for all child routines to be done.

The signature of the `coroutineScope` function is as follows:

```
suspend fun <R> coroutineScope(
    block: suspend CoroutineScope.() -> R
): R
```

The function therefore takes a lambda (with receiver `CoroutineScope`) that has no arguments and returns a generic value. The function is a suspending function, so it must be called from a suspending function or other coroutine.

A simple example of how to use `coroutineScope` is shown directly on the Kotlin home page and in Example 13-4.

Example 13-4. Using the `coroutineScope` builder

```
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

suspend fun main() = coroutineScope {          ❶
    for (i in 0 until 10) {
        launch {                               ❷
            delay(1000L - i * 10)              ❸
            print("❤$i ")
        }
    }
}
```

❶ `coroutineScope` builder

❷ Launches 10 coroutines

❸ Delays each one by a decreasing amount

The example launches 10 coroutines, each delayed by 10 milliseconds less than the previous one. In other words, the printed output contains a heart emoji and a number in descending order:

```
❤9 ❤8 ❤7 ❤6 ❤5 ❤4 ❤3 ❤2 ❤1 ❤0
```

This example shows a common pattern. In practice, start with `coroutineScope` to establish the scope of the included coroutines, and inside the resulting block you can use `launch` or `async` to handle individual tasks. The scope will then wait until all coroutines are completed before exiting, and if any of the coroutines fails, will also cancel the rest of them. This achieves a nice balance of control and error handling without having to poll to see whether routines are done and prevents leaks in case a routine fails.

The convention of running all coroutines inside `coroutineScope` to ensure that if one fails, all will be cancelled, is known as *structured concurrency*.

Coroutines can be confusing because there are so many moving parts and so many possible combinations. Fortunately, only a handful of combinations appear in practice, as shown in this recipe.

13.2 Replacing async/await with withContext

Problem

You want to simplify code that starts a coroutine with `async` and then immediately waits for it to complete with `await`.

Solution

Replace the combination of `async/await` with `withContext`.

Discussion

The `CoroutineScope` class also defines an extension function called `withContext`. Its signature is given by the following:

```kotlin
suspend fun <T> withContext(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T
```

The documentation says that `withContext` "calls the specified suspending block with a given coroutine context, suspends until it completes, and returns the result." In practice, use `withContext` to replace a combination of `async` with an immediate `await`, as in Example 13-5.

Example 13-5. Replacing `async` and `await` with `withContext`

```
suspend fun retrieve1(url: String) = coroutineScope {
    async(Dispatchers.IO) {
        println("Retrieving data on ${Thread.currentThread().name}")
        delay(100L)
        "asyncResults"
    }.await()
}

suspend fun retrieve2(url: String) =
    withContext(Dispatchers.IO) {
        println("Retrieving data on ${Thread.currentThread().name}")
        delay(100L)
        "withContextResults"
    }

fun main() = runBlocking<Unit> {
    val result1 = retrieve1("www.mysite.com")
    val result2 = retrieve2("www.mysite.com")
    println("printing result on ${Thread.currentThread().name} $result1")
    println("printing result on ${Thread.currentThread().name} $result2")
}
```

The `main` function starts with `runBlocking`, again typical of a simple demo like this. The two functions `retrieve1` and `retrieve2` do the same thing, which is to `delay` for 100 milliseconds and then return a string. The results are shown here (note the ordering could be different):

```
Retrieving data on DefaultDispatcher-worker-2
Retrieving data on DefaultDispatcher-worker-2
printing result on main withContextResults
printing result on main asyncResults
```

Both are using the `Dispatchers.IO` dispatcher (discussed in Recipe 13.3), so the only difference between the two functions is that one uses `async/await` and the other replaces it with `withContext`. In fact, when IntelliJ IDEA sees you using `async` with an immediate `await`, it will suggest replacing it with `withContext` as shown, and will do it for you if you let it.

13.3 Working with Dispatchers

Problem

You need to use a dedicated thread pool to do I/O or other tasks.

Solution

Use the proper dispatcher in the `Dispatchers` class.

Discussion

Coroutines execute in a context defined by a `CoroutineContext` type, which includes a coroutine dispatcher represented by an instance of the `CoroutineDispatcher` class. The dispatcher determines which thread or thread pool the coroutines use for their execution.

When you use a builder like `launch` or `async`, you can specify the dispatcher you want to use through an optional `CoroutineContext` parameter.

Built-in dispatchers provided by the library include the following:

- `Dispatchers.Default`
- `Dispatchers.IO`
- `Dispatchers.Unconfined`

The last one should not normally be used in application code.

The `Default` dispatcher uses a common pool of shared background threads. It is appropriate for coroutines that consume extensive amounts of computation resources.

The `IO` dispatcher uses a shared pool of on-demand created threads and is designed for offloading I/O-intensive blocking operations, like file I/O or blocking networking I/O.

Using either is quite simple. Just add them as an argument to `launch`, `async`, or `withContext` as needed. See Example 13-6.

Example 13-6. Using the `Default` and `I/O` dispatchers

```kotlin
fun main() = runBlocking<Unit> {
    launchWithIO()
    launchWithDefault()
}
```

```
suspend fun launchWithIO() {
    withContext(Dispatchers.IO) {              ❶
        delay(100L)
        println("Using Dispatchers.IO")
        println(Thread.currentThread().name)
    }
}

suspend fun launchWithDefault() {
    withContext(Dispatchers.Default) {         ❷
        delay(100L)
        println("Using Dispatchers.Default")
        println(Thread.currentThread().name)
    }
}
```

❶ `I/O` dispatcher

❷ `Default` dispatcher

The results are shown here (worker numbers may differ):

```
Using Dispatchers.IO
DefaultDispatcher-worker-3
Using Dispatchers.Default
DefaultDispatcher-worker-2
```

You can specify either dispatcher when the coroutines are launched.

Some tutorials refer to the functions `newSingleThreadContext` and `newFixedThreadPoolContext` as functions to create dispatchers. Both are considered obsolete and will be replaced in the future. To get similar functionality, use the `asCoroutineDispatcher` function on a Java `ExecutorService`, as described later in this recipe.

Android dispatchers

In addition to the dispatchers already discussed, the Android API includes a dispatcher called `Dispatchers.Main`. This is typical of UI toolkits, where you want to do all work updating the UI on `Main`, but any work that requires extra time or delays off of `Main`.

To get the Android `Main` dispatcher, you need to include the `kotlinx-coroutines-android` dependency. In a Gradle build file, that looks like this:

```
dependencies {
  implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:x.x.x"
  implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:x.x.x"
}
```

Here, the x.x.x values should be replaced by the latest version.

The Android components library makes additional life-cycle dispatchers available as well. See the details of the Android KTX library, specifically its lifecycle-viewmodel implementation. In fact, Android often recommends launching coroutines on viewModelScope, which is defined by that library.

See Also

Using a Java executor service as a source of coroutine dispatchers is discussed in Recipe 13.4. Android dispatchers are discussed further in Recipe 13.5.

13.4 Running Coroutines on a Java Thread Pool

Problem

You want to supply your own custom thread pool for coroutines to use.

Solution

Use the asCoroutineDispatcher function on Java's ExecutorService.

Discussion

The Kotlin library adds an extension method on java.util.concurrent.ExecutorService called asCoroutineDispatcher. As the documentation says, the function converts an instance of ExecutorService to an implementation of ExecutorCoroutineDispatcher.

To use it, use the Executors class to define your thread pool and then convert it to be used as a dispatcher, as in Example 13-7.

Example 13-7. Using a thread pool as a coroutine dispatcher

```
import kotlinx.coroutines.asCoroutineDispatcher
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.withContext
import java.util.concurrent.Executors

fun main() = runBlocking<Unit> {
    val dispatcher = Executors.newFixedThreadPool(10)    ❶
        .asCoroutineDispatcher()

    withContext(dispatcher) {                             ❷
        delay(100L)
        println(Thread.currentThread().name)
    }

    dispatcher.close()                                    ❸
}
```

❶ Creates a thread pool of size 10

❷ Uses the pool as a dispatcher for coroutines

❸ Shuts down the thread pool

The output prints `pool-1-thread-2`, indicating that the system chose to run the coroutine on thread 2 of pool 1.

Note the last line in that example, which invokes the `close` function on the dispatcher. This is necessary, because the executor service will continue to run without it, meaning the `main` function will never exit.

While the preceding technique works, it's also an interesting illustration of how Kotlin goes about solving this sort of problem. Normally, to get a Java `ExecutorService` to stop, you invoke the `shutdown` or `shutdownNow` method. Therefore, in principle you could rewrite the example to keep a reference to the `ExecutorService` and shut it down manually, as in Example 13-8.

Example 13-8. Shutting down the thread pool

```kotlin
val pool = ExecutorService.newFixedThreadPool(10)
withContext(pool.asCoroutineDispatcher()) {
    // ... same as before ...
}
pool.shutdown()
```

The problem with that approach is that a user might forget to call the `shutdown` method. Java solves problems like that by implementing the `AutoCloseable` interface with a `close` method, so that you can wrap the code in a try-with-resources block. Unfortunately, the method you want to call here is `shutdown`, not `close`.

The developers of the Kotlin library therefore made a change to the underlying `ExecutorCoroutineDispatcher` class, an instance of which is created in the preceding code. They refactored it to implement the `Closeable` interface, so that the new abstract class is called `CloseableCoroutineDispatcher`, whose `close` method looks like this:

```kotlin
import java.util.concurrent.ExecutorService

abstract class ExecutorCoroutineDispatcher: CoroutineDispatcher(), Closeable {
    abstract override fun close()
    abstract val executor: Executor
}

// Then, in subclasses:
override fun close()  {
    (executor as? ExecutorService)?.shutdown()
}
```

That means that the dispatchers created using the executor service now have a `close` function that will shut down the underlying executor service. The question then is how do you ensure that the `close` function is called, given that Kotlin doesn't support a try-with-resources construct similar to Java? What Kotlin does have is a `use` function. The definition of `use` is shown in Example 13-9.

Example 13-9. The `use` function

```
inline fun <T : Closeable?, R> T.use(block: (T) -> R): R
```

Therefore, `use` is defined as an extension function on Java's `Closeable` interface. That gives a straightforward solution to the problem of shutting down the Java executor service, shown in Example 13-10.

Example 13-10. Closing the dispatcher with `use`

```
Executors.newFixedThreadPool(10).asCoroutineDispatcher().use {
    withContext(it) {
        delay(100L)
        println(Thread.currentThread().name)
    }
}
```

This will close the dispatcher when the `use` block ends, which will also close the underlying thread pool.

See Also

The `use` function is described in Recipe 10.1.

13.5 Cancelling Coroutines

Problem

You need to cancel an asynchronous process running in a coroutine.

Solution

Use the `Job` reference returned by the `launch` builder, or one of the functions such as `withTimeout` or `withTimeoutOrNull`.

Discussion

The `launch` builder returns an instance of type `Job`, which can be used to cancel coroutines. Example 13-11 is based on an example from the Kotlin reference guide.

Example 13-11. Cancelling a job

```kotlin
fun main() = runBlocking {
    val job = launch {
        repeat(100) { i ->
            println("job: I'm waiting $i...")
            delay(100L)
        }
    }
    delay(500L)
    println("main: That's enough waiting")
    job.cancel()
    job.join()
    println("main: Done")
}
```

The `launch` builder returns an instance of `Job`, which is assigned to a local variable. Then 100 coroutines are launched using the `repeat` function.

Outside the `launch` block, the main function gets tired of waiting for them all, so it cancels the job. The `join` function waits for the job to be completed, and then the program exits. The output from the program is as follows:

```
job: I'm waiting 0...
job: I'm waiting 1...
job: I'm waiting 2...
job: I'm waiting 3...
job: I'm waiting 4...
main: That's enough waiting
main: Done
```

There is also a `cancelAndJoin` function that combines `cancel` and `join` calls.

If the reason you want to cancel a job is that it might be taking too long, you can also use the `withTimeout` function. The signature for `withTimeout` is shown here:

```
suspend fun <T> withTimeout(
    timeMillis: Long,
    block: suspend CoroutineScope.() -> T
): T
```

The function runs a suspending block of code inside a coroutine and throws a `TimeoutCancellationException` if the timeout is exceeded. An example of its use, again based on an example from the reference manual, is given in Example 13-12.

Example 13-12. Using `withTimeout`

```
fun main() = runBlocking {
    withTimeout(1000L) {
        repeat(50) { i ->
            println("job: I'm waiting $i...")
            delay(100L)
        }
    }
}
```

The result now is as follows:

```
job: I'm waiting 0...
job: I'm waiting 1...
job: I'm waiting 2...
job: I'm waiting 3...
job: I'm waiting 4...
job: I'm waiting 5...
job: I'm waiting 6...
job: I'm waiting 7...
job: I'm waiting 8...
job: I'm waiting 9...
Exception in thread "main" kotlinx.coroutines.TimeoutCancellationException:
    Timed out waiting for 1000 ms
at kotlinx.coroutines.TimeoutKt.TimeoutCancellationException(Timeout.kt:126)
    // ... rest of stack trace ...
```

You can catch the exception if you want, or use `withTimeoutOrNull`, which returns a `null` on timeout rather than throwing an exception.

Cancelling jobs in Android

Android provides an additional dispatcher called `Dispatchers.Main`, which operates on the UI thread. A common implementation pattern is to make the `MainActivity` implement `CoroutineScope`, provide a context when needed, and then close it if necessary. The technique is shown in Example 13-13.

Example 13-13. Using dispatchers in Android

```
class MainActivity : AppCompatActivity(), CoroutineScope {
    override val coroutineContext: CoroutineContext    ❶
        get() = Dispatchers.Main + job

    private lateinit var job: Job                      ❷

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        job = Job()                                    ❸
    }

    override fun onDestroy() {
        job.cancel()                                   ❹
        super.onDestroy()
    }
}
```

❶ Creates the context using the overloaded plus operator

❷ Initializes a property when ready

❸ Now it's ready

❹ Cancels the job if the activity is being destroyed

The use of the late initialized variable `job` provides access to it in case of cancellation. To do the work, now simply launch coroutines as necessary, as in Example 13-14.

Example 13-14. Launching coroutines from Android

```
fun displayData() {
    launch {                                     ❶
        val data = async(Dispatchers.IO) {       ❷
            // ... get data over network ...
        }
        updateDisplay(data.await())              ❸
    }
}
```

❶ Launches using the `coroutineContext` property

❷ Switches to `Dispatchers.IO` for networked call

❸ Back to `Dispatchers.Main` to update the UI

As soon as the activity is destroyed, the task will get cancelled as well.

Recent versions of Android architecture components provide additional scopes, like `viewModelScope`, that automatically cancel a job when the `ViewModel` is cleared. This is part of the Android KTX library, so you need to add the proper dependency to your build:

```
dependencies {
  // ... as before ...
  implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:x.x.x"
}
```

This adds the `viewModelScope` property, which you can use to launch coroutines on any dispatcher.

13.6 Debugging Coroutines

Problem

You need more information about executing coroutines.

Solution

On the JVM, run with the `-Dkotlinx.coroutines.debug` flag.

Discussion

Debugging asynchronous programs is always difficult, because multiple operations can be running at the same time. Fortunately, the coroutines library includes a simple debugging feature.

To execute coroutines in debug mode (on the JVM), use the system property `kotlinx.coroutines.debug`.

Alternatively, you can enable debugging with the `-ea` (enable assertions) flag on the Java command line.

Debug mode attaches a unique name to every launched coroutine. Example 13-5, reproduced here for convenience, shows two coroutines being launched in addition to the main thread:

```
suspend fun retrieve1(url: String) = coroutineScope {
    async(Dispatchers.IO) {
        println("Retrieving data on ${Thread.currentThread().name}")
        delay(100L)
        "asyncResults"
    }.await()
}

suspend fun retrieve2(url: String) =
    withContext(Dispatchers.IO) {
        println("Retrieving data on ${Thread.currentThread().name}")
        delay(100L)
        "withContextResults"
    }

fun main() = runBlocking<Unit> {
    val result1 = retrieve1("www.mysite.com")
    val result2 = retrieve2("www.mysite.com")
    println("printing result on ${Thread.currentThread().name} $result1")
    println("printing result on ${Thread.currentThread().name} $result2")
}
```

If you execute this program with `-Dkotlinx.coroutines.debug`, the output is as follows:

```
Retrieving data on DefaultDispatcher-worker-1 @coroutine#1
Retrieving data on DefaultDispatcher-worker-1 @coroutine#2
printing result on main @coroutine#1 withContextResults
printing result on main @coroutine#1 asyncResults
```

Each coroutine has been given a unique name (`@coroutine#1`, etc.) that is displayed as part of the thread name.

While this is helpful, sometimes you want to supply names to the coroutines rather than use the generated ones. The Kotlin library includes a class called `CoroutineName` for this purpose. The `CoroutineName` constructor produces a context element that can be used as the thread name, as in Example 13-15.

Example 13-15. Naming the coroutines

```
suspend fun retrieve1(url: String) = coroutineScope {
    async(Dispatchers.IO + CoroutineName("async")) {  ❶
        // ... as before ...
    }.await()
}

suspend fun retrieve2(url: String) =
    withContext(Dispatchers.IO + CoroutineName("withContext")) { (1)
        // ... as before ...
    }
```

(1) Adds (literally) a coroutine name

The result now looks like this:

```
Retrieving data on DefaultDispatcher-worker-1 @withContext#1
Retrieving data on DefaultDispatcher-worker-1 @async#2
printing result on main @coroutine#1 withContextResults
printing result on main @coroutine#1 asyncResults
```

The words "async" and "withContext" now appear as the names of the coroutines. This is also a nice example of the overloaded plus operator when used on `CoroutineContext`. Another example of the plus operator being used is shown in Recipe 13.5 for Android applications.

Index

About the Author

Ken Kousen is a Java Champion, Oracle Developer Champion, and Grails Rock Star whose books include *Modern Java Recipes*, *Gradle Recipes for Android*, and *Making Java Groovy*. He is also a JetBrains Certified Kotlin Training Partner.

Colophon

The animal on the cover of *Kotlin Cookbook* is a kinkajou (*Potos flavus*), a long-tailed mammal native to Central and South America.

Kinkajous have golden-brown fur, small slightly webbed paws, and large black eyes—the latter of which assist this nocturnal creature in seeing better at night. Reversible hind feet and a prehensile, or gripping, tail aid the kinkajou's arboreal (treetop-dwelling) lifestyle. Although similar in appearance and behavior to primates, kinkajous evolved separately from primates and are most closely related to raccoons.

While technically an omnivore, most of the kinkajou's diet consists of fruit—particularly figs. Occasionally, they will dine on small vertebrates or bird eggs. Their curiously long tongue allows them to indulge in insects or nectar, earning these creatures the nickname "honey bear." In fact, captive populations are indeed fed honey, much to their delight.

A troop of kinkajous participates in social grooming and nesting. They can be aggressive when startled or awoken during the day, attacking with sharp teeth and claws, or emitting loud screeches and barks that echo through the surrounding canopy.

While the kinkajou's conservation status is currently listed as of Least Concern, deforestation is a potential threat to the kinkajou population. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Lydekker's Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.