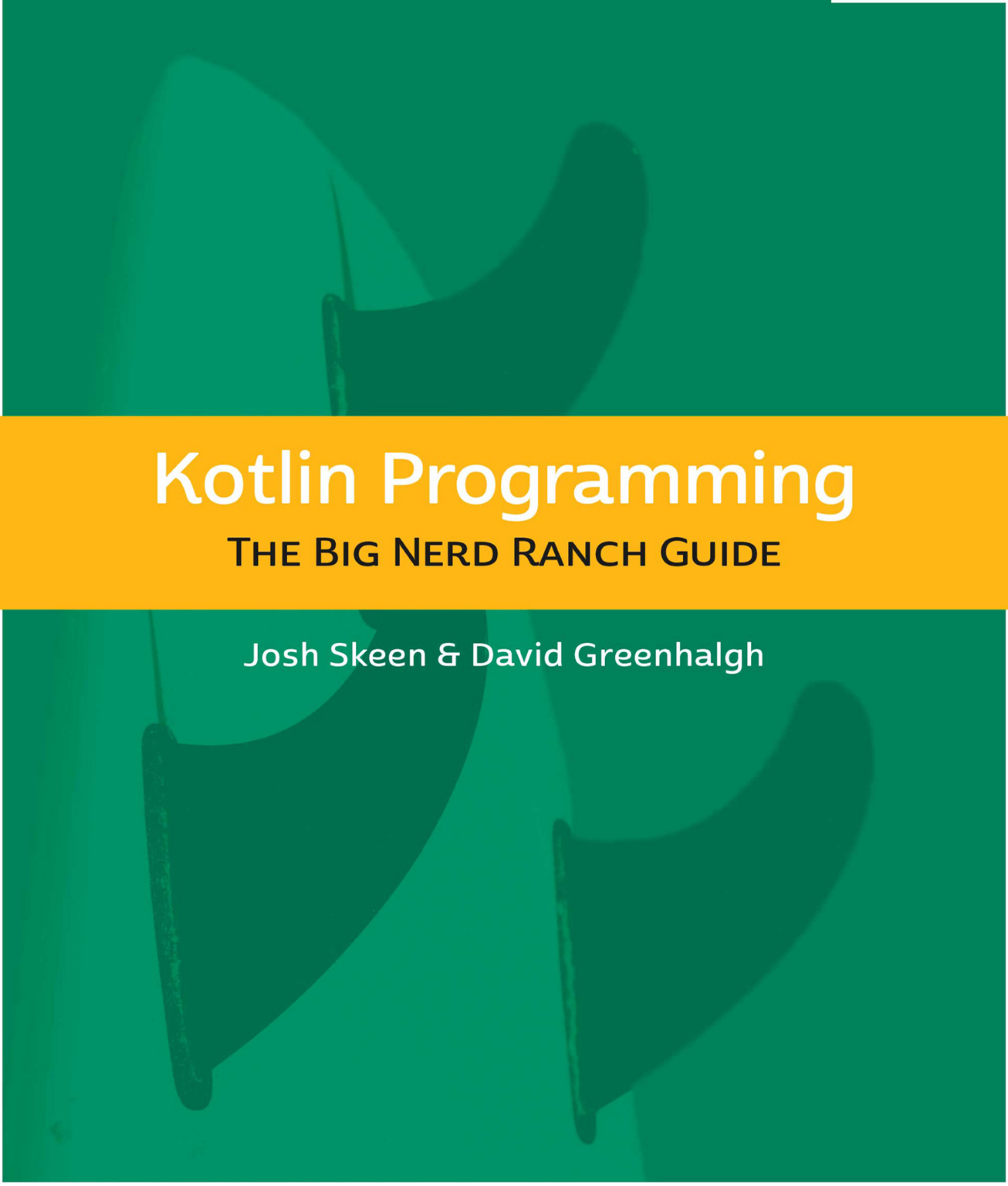
Big Nerd Ranch



Kotlin Programming

THE BIG NERD RANCH GUIDE

Josh Skeen & David Greenhalgh

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Kotlin Programming: The Big Nerd Ranch Guide

by Josh Skeen and David Greenhalgh

Copyright © 2018 Big Nerd Ranch, LLC

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

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

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

Exclusive worldwide distribution of the English edition of this book by

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

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

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

ISBN-10 0135162386
ISBN-13 978-0135162385

First edition, first printing, July 2018
Release D.1.1.1

Dedication

For Baker, the best little bug.

— J.S.

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

— D.G.

Acknowledgments

We received a lot of help in writing this book. Without that help, this book would not be what it is, and it may never even have happened. Thanks are due.

First, we need to say thank you to our colleagues at Big Nerd Ranch. Thank you to Stacy Henry and Aaron Hillegass for providing us with the time and space to write this book. It has been immensely gratifying to learn and teach Kotlin. We hope that this book lives up to the trust and the support that we have received.

Particular thanks are also due to our colleagues at Big Nerd Ranch. Your careful teaching revealed many bugs in the text, and your thoughtful recommendations led to many improvements in our approach. It is truly wonderful to have colleagues such as you. Thank you Kristin Marsicano, Bolot Kerimbaev, Brian Gardner, Chris Stewart, Paul Turner, Chris Hare, Mark Allison, Andrew Lunsford, Rafael Moreno Cesar, Eric Maxwell, Andrew Bailey, Jeremy Sherman, Christian Keur, Mikey Ward, Steve Sparks, Mark Dalrymple, CBQ, and everyone else at the Ranch who helped us with this work.

Our colleagues in operations, marketing, and sales are also instrumental. Classes would literally never be scheduled without their work. Thank you Heather Sharpe, Mat Jackson, Rodrigo "Ram Rod" Perez-Velasco, Nicholas Stolte, Justin Williams, Dan Barker, Israel Machovec, Emily Herman, Patrick Freeman, Ian Eze, and Nikki Porter. We cannot do what we do without what you do.

Special thanks and an extra bit of karma are also owed to our amazing students who were adventurous enough to join us for the early access version of the course and were kind enough to help us identify errata. Without your feedback and insights into how to improve the course, this text would not be where it is today. Those students include: Santosh Katta, Abdul Hannan, Chandra Mohan, Benjamin DiGregorio, Peng Wan, Kapil Bhalla, Girish Hanchinal, Hashan Godakanda, Mithun Mahadevan, Brittany Berlanga, Natalie Ryan, Balarka Velidi, Pranay Airan, Jacob Rogers, Jean-Luc Delpech, Dennis Lin, Kristina Thai, Reid Baker, Setareh Lotfi, Harish Ravichandran, Matthew Knapp, Nathan Klee, Brian Lee, Heidi Muth, Martin Davidsson, Misha Burshteyn, Kyle Summers, Cameron Hill, Vidhi Shah, Fabrice Di Meglio, Jared Burrows, Riley Brewer, Michael Krause, Tyler Holland, Gajendra Singh, Pedro Sanchez, Joe Cyboski, Zach Waldowski, Noe Arzate, Allan Caine, Zack Simon, Josh Meyers, Rick Meyers, Stephanie Guevara, Abdulrahman Alshmrani, Robert Edwards, Maribel Montejano, and Mohammad Yusuf.

We want to extend a special thank you to our colleagues and members of the Android community who helped us test the book's accuracy, clarity, and ease of use. Without your external perspective, putting this book together would have been even more daunting. Thank you Jon Reeve, Bill Phillips, Matthew Compton, Vishnu Rajeevan, Scott Stanlick, Alex Lumans, Shauvik Choudhary, and Jason Atwood.

We also need to acknowledge the many talented folks who worked on the book with us. Elizabeth Holaday, our editor, helped refine the book, crystallize its strengths, and diminish its weaknesses. Anna Bentley, our copyeditor, found and corrected errors and ultimately made us look smarter than we are. Ellie Volckhausen designed the cover. And Chris Loper designed and produced the print book and the EPUB and Kindle versions.

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

Table of Contents

Introducing Kotlin xiii
- Why Kotlin? xiii
- Who Is This Book For? xiii
- How to Use This Book xiv
  - For the More Curious xiv
  - Challenges xiv
  - Typographical conventions xiv
- Looking Forward xv

1. Your First Kotlin Application 1
- Installing IntelliJ IDEA 1
- Your First Kotlin Project 2
  - Creating your first Kotlin file 6
  - Running your Kotlin file 8
- The Kotlin REPL 10
- For the More Curious: Why Use IntelliJ? 11
- For the More Curious: Targeting the JVM 12
- Challenge: REPL Arithmetic 12

2. Variables, Constants, and Types 13
- Types 13
- Declaring a Variable 14
- Kotlin’s Built-In Types 16
- Read-Only Variables 17
- Type Inference 19
- Compile-Time Constants 21
- Inspecting Kotlin Bytecode 22
- For the More Curious: Java Primitive Types in Kotlin 25
- Challenge: hasSteed 26
- Challenge: The Unicorn’s Horn 26
- Challenge: Magic Mirror 26

3. Conditionals 27
- if/else Statements 27
  - Adding more conditions 30
  - Nested if/else statements 31
  - More elegant conditionals 32
- Ranges 37
- when Expressions 38
- String Templates 40
- Challenge: Trying Out Some Ranges 41
- Challenge: Enhancing the Aura 42
- Challenge: Configurable Status Format 42

4. Functions 43
- Extracting Code to Functions 43
- Anatomy of a Function 46
  - Function header 46
  - Function body 49

Function scope 50
Calling a Function 51
Refactoring to Functions 52
Writing Your Own Functions 54
Default Arguments 56
Single-Expression Functions 57
Unit Functions 58
Named Function Arguments 59
For the More Curious: The Nothing Type 60
For the More Curious: File-Level Functions in Java 61
For the More Curious: Function Overloading 62
For the More Curious: Function Names in Backticks 63
Challenge: Single-Expression Functions 64
Challenge: Fireball Inebriation Level 64
Challenge: Inebriation Status 64
5. Anonymous Functions and the Function Type 67
Anonymous Functions 67
The function type 69
Implicit returns 70
Function arguments 71
The it keyword 72
Accepting multiple arguments 73
Type Inference Support 74
Defining a Function That Accepts a Function 75
Shorthand syntax 76
Function Inlining 77
Function References 78
Function Type as Return Type 80
For the More Curious: Kotlin's Lambdas Are Closures 81
For the More Curious: Lambdas vs Anonymous Inner Classes 82
6. Null Safety and Exceptions 83
Nullability 83
Kotlin's Explicit Null Type 85
Compile Time vs Runtime 86
Null Safety 87
Option one: the safe call operator 88
Option two: the double-bang operator 89
Option three: checking whether a value is null with if 90
Exceptions 92
Throwing an exception 94
Custom exceptions 95
Handling exceptions 96
Preconditions 97
Null: What Is It Good For? 99
For the More Curious: Checked vs Unchecked Exceptions 100
For the More Curious: How Is Nullability Enforced? 101
7. Strings 103
Extracting Substrings 103

substring 103
split 106
String Manipulation 107
Strings are immutable 109
String Comparison 109
For the More Curious: Unicode 111
For the More Curious: Traversing a String's Characters 112
Challenge: Improving DragonSpeak 112
8. Numbers 113
Numeric Types 113
Integers 114
Decimal Numbers 116
Converting a String to a Numeric Type 116
Converting an Int to a Double 117
Formatting a Double 118
Converting a Double to an Int 119
For the More Curious: Bit Manipulation 121
Challenge: Remaining Pints 122
Challenge: Handling a Negative Balance 122
Challenge: Dragoncoin 122
9. Standard Functions 123
apply 123
let 124
run 125
with 126
also 126
takeIf 127
takeUnless 127
Using Standard Library Functions 128
10. Lists and Sets 129
Lists 129
Accessing a list's elements 130
Changing a list's contents 133
Iteration 137
Reading a File into a List 140
Destructuring 142
Sets 142
Creating a set 143
Adding elements to a set 144
while Loops 146
The break Expression 148
Collection Conversion 149
For the More Curious: Array Types 150
For the More Curious: Read-Only vs Immutable 151
Challenge: Formatted Tavern Menu 152
Challenge: Advanced Formatted Tavern Menu 152
11. Maps 153
Creating a Map 153

Accessing Map Values 155
Adding Entries to a Map 156
Modifying Map Values 158
Challenge: Tavern Bouncer 162
12. Defining Classes 163
Defining a Class 163
Constructing Instances 164
Class Functions 164
Visibility and Encapsulation 166
Class Properties 167
Property getters and setters 168
Property visibility 171
Computed properties 172
Refactoring NyetHack 173
Using Packages 181
For the More Curious: A Closer Look at var and val Properties 182
For the More Curious: Guarding Against Race Conditions 184
For the More Curious: Package Private 185
13. Initialization 187
Constructors 187
Primary constructors 188
Defining properties in a primary constructor 189
Secondary constructors 189
Default arguments 191
Named arguments 192
Initializer Blocks 193
Property Initialization 194
Initialization Order 196
Delaying Initialization 197
Late initialization 198
Lazy initialization 198
For the More Curious: Initialization Gotchas 200
Challenge: The Riddle of Excalibur 202
14. Inheritance 205
Defining the Room Class 205
Creating a Subclass 206
Type Checking 212
The Kotlin Type Hierarchy 214
Type casting 215
Smart casting 216
For the More Curious: Any 217
15. Objects 219
The object Keyword 219
Object declarations 220
Object expressions 224
Companion objects 225
Nested Classes 226
Data Classes 228

toString ... 229
equals ... 230
copy ... 230
Destructuring declarations ... 230
Enumerated Classes ... 231
Operator Overloading ... 233
Exploring the World of NyetHack ... 235
For the More Curious: Defining Structural Comparison ... 238
For the More Curious: Algebraic Data Types ... 240
Challenge: “Quit” Command ... 242
Challenge: Implementing a World Map ... 242
Challenge: Ring the Bell ... 242
16. Interfaces and Abstract Classes ... 243
Defining an Interface ... 243
Implementing an Interface ... 244
Default Implementations ... 248
Abstract Classes ... 248
Combat in NyetHack ... 251
17. Generics ... 255
Defining Generic Types ... 255
Generic Functions ... 256
Multiple Generic Type Parameters ... 258
Generic Constraints ... 259
vararg and get ... 260
in and out ... 263
For the More Curious: The reified Keyword ... 267
18. Extensions ... 269
Defining Extension Functions ... 270
Defining an extension on a superclass ... 271
Generic Extension Functions ... 272
Extension Properties ... 274
Extensions on Nullable Types ... 275
Extensions, Under the Hood ... 276
Extracting to Extensions ... 277
Defining an Extensions File ... 279
Renaming an Extension ... 281
Extensions in the Kotlin Standard Library ... 282
For the More Curious: Function Literals with Receivers ... 283
Challenge: toDragonSpeak Extension ... 284
Challenge: Frame Extension ... 284
19. Functional Programming Basics ... 285
Function Categories ... 285
Transforms ... 285
Filters ... 287
Combines ... 288
Why Functional Programming? ... 289
Sequences ... 290
For the More Curious: Profiling ... 292

For the More Curious: Arrow.kt 292
Challenge: Reversing the Values in a Map 293
Challenge: Applying Functional Programming to Tavern.kt 293
Challenge: Sliding Window 294
20. Java Interoperability 295
Interoperating with a Java Class 295
Interoperability and Nullity 296
Type Mapping 299
Getters, Setters, and Interoperability 300
Beyond Classes 302
Exceptions and Interoperability 310
Function Types in Java 313
21. Building Your First Android Application with Kotlin 315
Android Studio 315
Gradle configuration 321
Project organization 324
Defining a UI 324
Running the App on an Emulator 327
Generating a Character 329
The Activity Class 330
Wiring Up Views 330
Kotlin Android Extensions Synthetic Properties 333
Setting a Click Listener 335
Saved Instance State 336
Reading from the saved instance state 339
Refactoring to an Extension 340
For the More Curious: Android KTX and Anko Libraries 341
22. Introduction to Coroutines 343
Parsing Character Data 343
Fetching Live Data 346
The Android Main Thread 348
Enabling Coroutines 349
Specifying a Coroutine with async 349
launch vs async/await 351
Suspending Functions 351
Challenge: Live Data 352
Challenge: Minimum Strength 352
23. Afterword 353
Where to Go from Here 353
Shameless Plugs 353
Thank You 353
Appendix: More Challenges 357
Leveling Up with Exercism 357
Glossary 363
Index 373

Introducing Kotlin

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

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

Why Kotlin?

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

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

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

And Kotlin is not just a better language to write code to run on the Java Virtual Machine. It is a multiplatform language that aims to be general purpose: Kotlin can be used to write native macOS and Windows applications, JavaScript applications, and, of course, Android applications. Platform independence means that Kotlin has a wide variety of uses.

Who Is This Book For?

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

Android support might be why you are reading this book, but the book is not limited to Kotlin programming for Android. In fact, except in one advanced chapter, Chapter 21, all the Kotlin code in this book is agnostic to the Android framework. That said, if you are interested in using Kotlin for Android application development, this book shows off some common patterns that make writing Android apps a breeze in Kotlin.

Although Kotlin has been influenced by a number of other languages, you do not need to know the ins and outs of any other language to learn Kotlin. From time to time, we will discuss the Java code equivalent for Kotlin code you have written. If you have Java experience, this will help you understand the relationship between the two languages. If you do not know Java, seeing how another language tackles the same problems can help you grasp the principles that have shaped Kotlin's development.

How to Use This Book

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

Also, each chapter builds on the topics presented in the last, so we recommend that you do not jump around. Even if you feel that you are familiar with a topic in other languages, we suggest that you read straight through – Kotlin handles many problems in unique ways. You will begin with introductory topics like variables and lists, work your way through object-oriented and functional programming techniques, and understand along the way what makes Kotlin such a powerful language. By the end of the book, you will have built your knowledge of Kotlin from that of a beginner to a more advanced developer.

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

For the More Curious

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

Challenges

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

Typographical conventions

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

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

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

~~~
var x = "Python"
~~var y = "Java"~~
**var z = "Kotlin"**
~~~

Kotlin is a relatively young language, so many coding conventions are still being figured out. Over time, you will likely develop your own style, but we tend to adhere to JetBrains' and Google's Kotlin style guides:

- JetBrains' coding conventions: `kotlinlang.org/docs/reference/coding-conventions.html`
- Google's style guide, including conventions for Android code and interoperability: `android.github.io/kotlin-guides/style.html`

Looking Forward

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

1

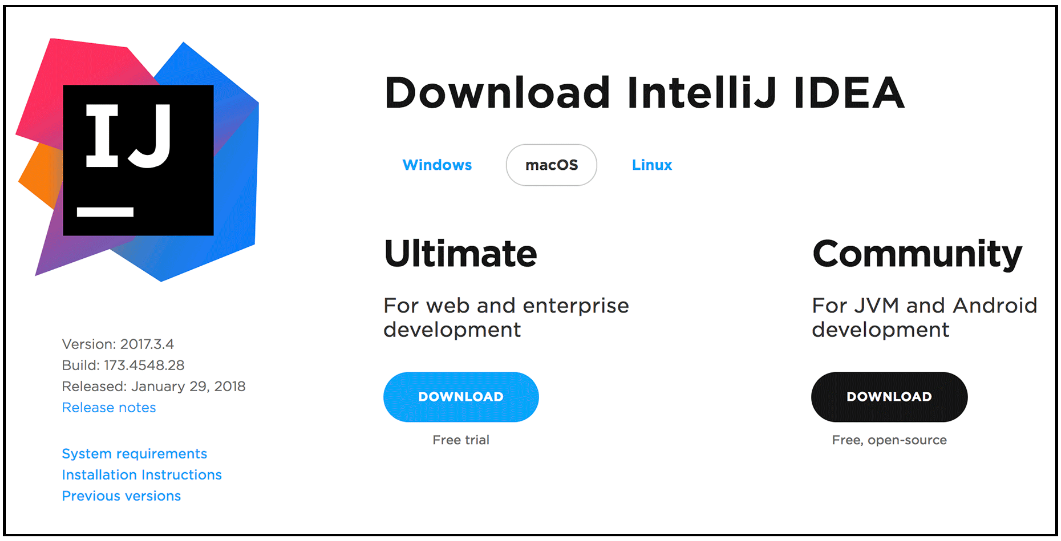
Your First Kotlin Application

In this chapter you will write your first Kotlin program, using IntelliJ IDEA. While completing this programming rite of passage, you will familiarize yourself with your development environment, create a new Kotlin project, write and run Kotlin code, and inspect the resulting output. The project you create in this chapter will serve as a sandbox to easily try out new concepts you will encounter throughout this book.

Installing IntelliJ IDEA

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

Figure 1.1 Downloading IntelliJ IDEA Community Edition



Once it has downloaded, follow the installation instructions for your platform as described on the JetBrains installation and setup page at `jetbrains.com/help/idea/install-and-set-up-product.html`.

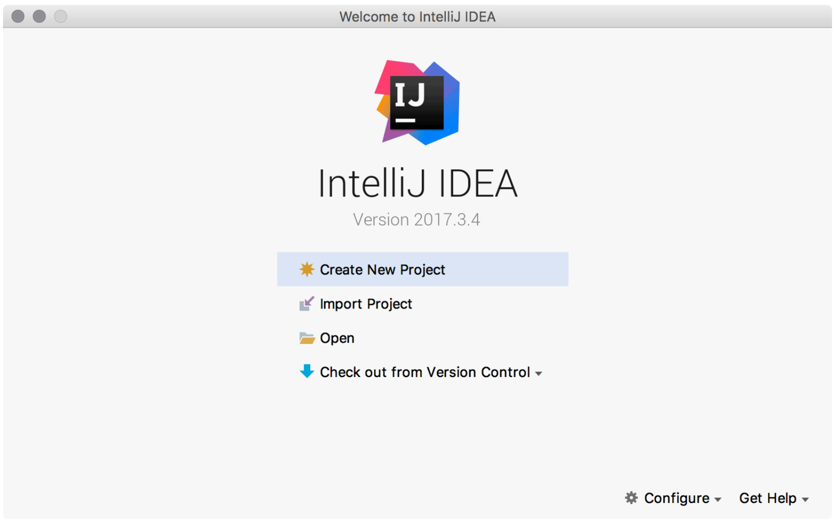
IntelliJ IDEA, called IntelliJ for short, helps you write well-formed Kotlin code. It also streamlines the development process with built-in tools for running, debugging, inspecting, and refactoring your code. You can read more about why we recommend IntelliJ for writing Kotlin code in the section called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

Your First Kotlin Project

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

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

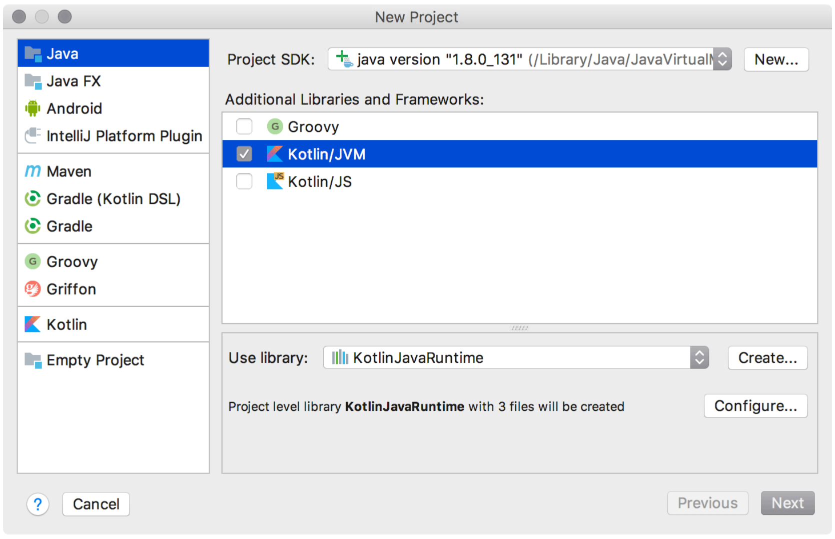
Figure 1.2 Welcome dialog



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

Click Create New Project. IntelliJ will display the New Project dialog, as shown in Figure 1.3.

Figure 1.3 New Project dialog



Project SDK: java version "1.8.0_131" (/Library/Java/JavaVirtualM

Additional Libraries and Frameworks:

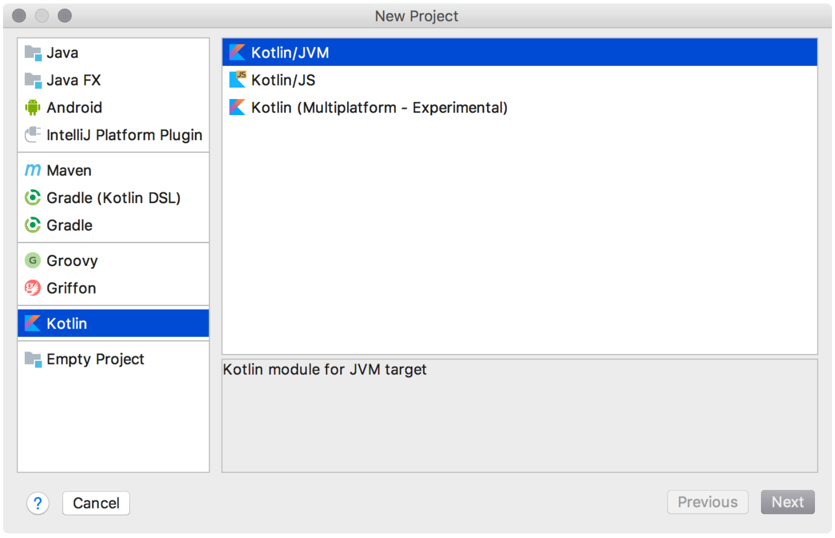
- [ ] Groovy
- [✓] Kotlin/JVM
- [ ] Kotlin/JS

Use library: KotlinJavaRuntime

Project level library **KotlinJavaRuntime** with 3 files will be created

In the New Project dialog, select Kotlin on the left and Kotlin/JVM on the right, as shown in Figure 1.4.

Figure 1.4 Creating a Kotlin/JVM project



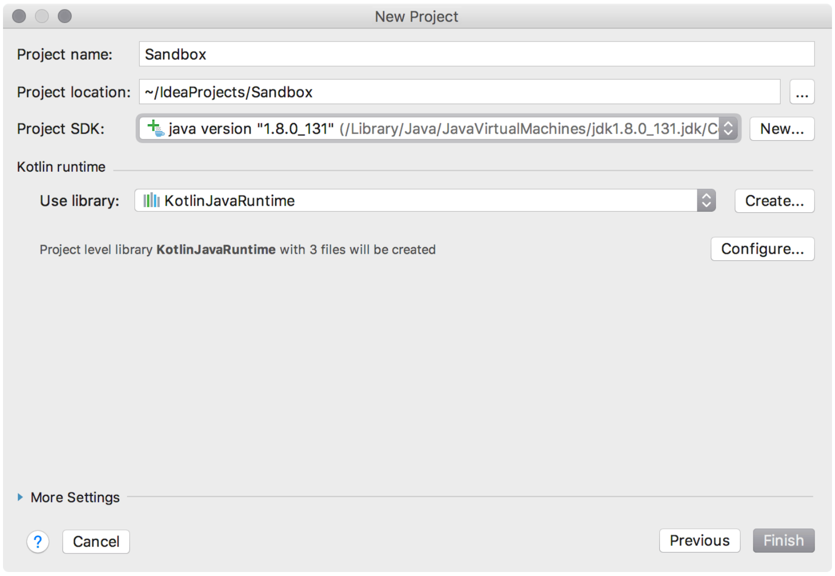
Kotlin module for JVM target

You can use IntelliJ to write code in languages other than Kotlin, including Java, Python, Scala, and Groovy. Selecting Kotlin/JVM tells IntelliJ you intend to use Kotlin. More specifically, Kotlin/JVM tells IntelliJ you intend to write Kotlin code that *targets*, or runs on, the Java Virtual Machine. One of the benefits of Kotlin is that it features a toolchain that allows you to write Kotlin code that can run on different operating systems and platforms.

(From here on, we will refer to the Java Virtual Machine as just “JVM,” as it is commonly called in the Java developer community. You can learn more about targeting the JVM in the section called *For the More Curious: Targeting the JVM* near the end of this chapter.)

Click Next in the New Project dialog. IntelliJ will display a dialog where you can choose settings for your new project (Figure 1.5). For the Project name, enter “Sandbox.” The Project location field will auto-populate. You can leave the location as is or select a new location by pressing the ... button to the right of the field. Select a Java 1.8 version from the Project SDK dropdown to link your project to Java Development Kit (JDK) version 8.

Figure 1.5 Naming the project



Project name: Sandbox
Project location: ~/IdeaProjects/Sandbox
Project SDK: java version "1.8.0_131" (/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/C
Kotlin runtime
Use library: KotlinJavaRuntime
Project level library **KotlinJavaRuntime** with 3 files will be created

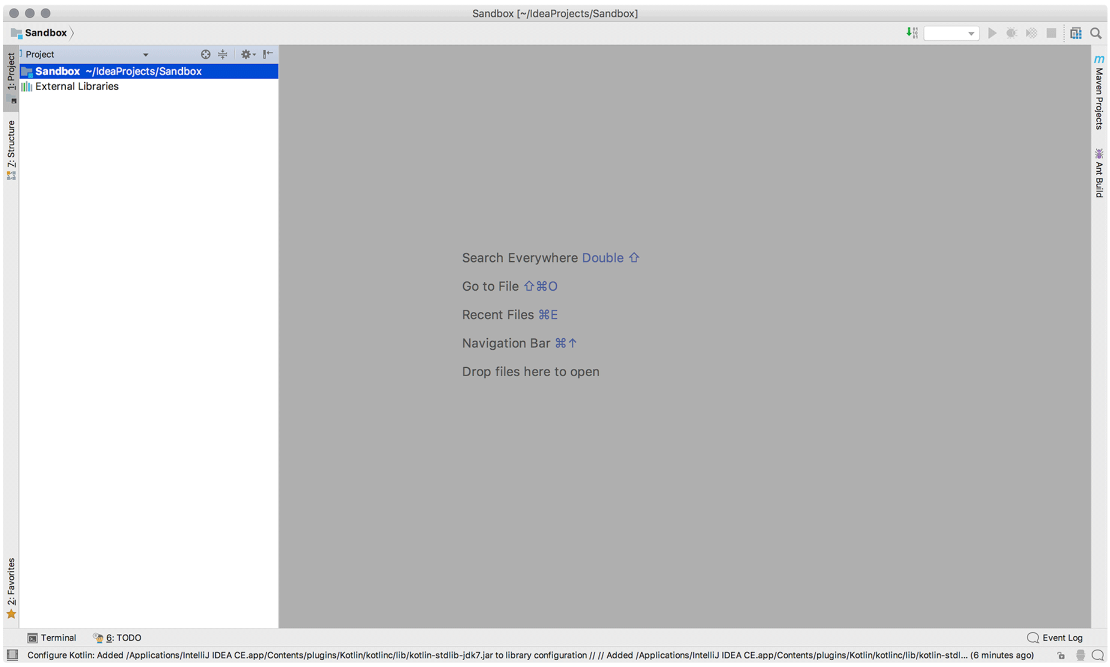
Why do you need the JDK to write a Kotlin program? The JDK gives IntelliJ access to the JVM and to Java tools that are necessary for converting your Kotlin code to bytecode (more on that in a moment). Technically, any version 6 or greater will work. But our experience, as of this writing, is that JDK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

If you do not see some version of Java 1.8 listed in the Project SDK dropdown, this means you have not yet installed JDK 8. Do so now before proceeding: Download JDK 8 for your specific platform from `oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`. Install the JDK, then restart IntelliJ. Work back through the steps outlined to this point to create a new project.

When your settings dialog looks like Figure 1.5, click Finish.

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

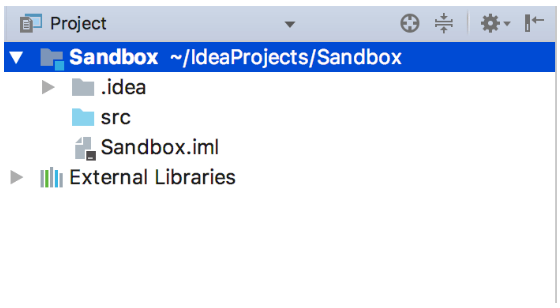
Figure 1.6 Default two-pane view



Configure Kotlin: Added /Applications/IntelliJ IDEA CE.app/Contents/plugins/Kotlin/kotlinc/lib/kotlin-stdlib-jdk7.jar to library configuration // // Added /Applications/IntelliJ IDEA CE.app/Contents/plugins/Kotlin/kotlinc/lib/kotlin-stdl... (6 minutes ago)

The pane on the left shows the *project tool window*. The pane on the right is currently empty. This is where you will view and edit the contents of your Kotlin files in the *editor*. Turn your attention to the project tool window on the left. Click the disclosure arrow to the left of the project name, Sandbox. It will expand to display the files contained in the project, as shown in Figure 1.7.

Figure 1.7 Project view



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

The `Sandbox.iml` file contains configuration information specific to your single module. The `.idea` folder contains settings files for the entire project as well as those specific to your interaction with the project in the IDE (for example, which files you have open in the editor). Leave these auto-generated files as they are.

The External Libraries entry contains information about libraries the project depends on. If you expand this entry you will see that IntelliJ automatically added Java 1.8 and KotlinJavaRuntime as dependencies for your project.

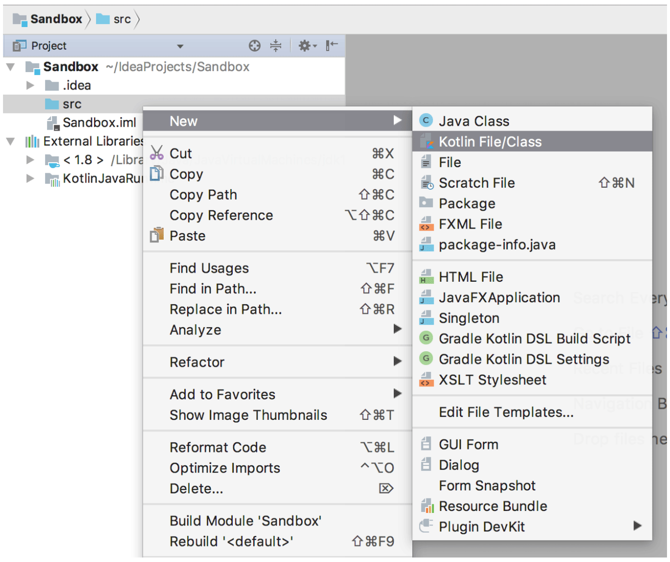
(You can learn more about IntelliJ project structure on the JetBrains documentation website at `jetbrains.org/intellij/sdk/docs/basics/project_structure.html`.)

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

Creating your first Kotlin file

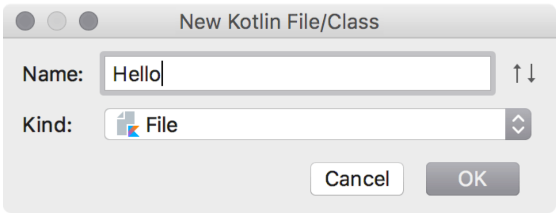
Right-click on the `src` folder in the project tool window. Select New and then Kotlin File/Class from the menu that appears (Figure 1.8).

Figure 1.8 Creating a new Kotlin file



In the New Kotlin File/Class dialog, type “Hello” in the Name field and leave the Kind field set to File (Figure 1.9).

Figure 1.9 Naming the file

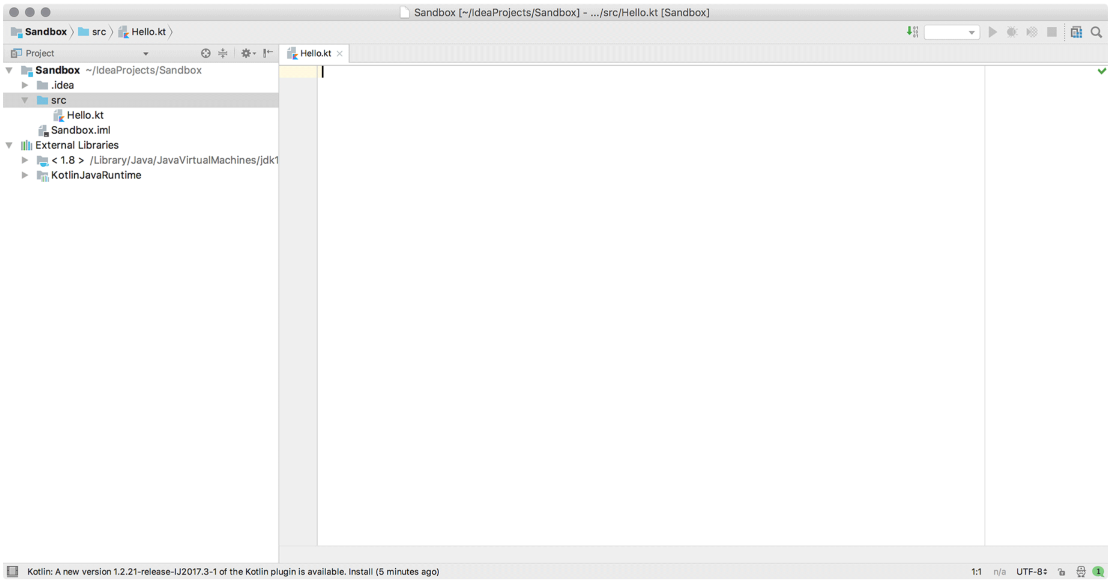


Name: Hello

Kind: File

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

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


Sandbox [~/IdeaProjects/Sandbox] - .../src/Hello.kt [Sandbox]
Sandbox
src
Hello.kt
Project
Hello.kt
Sandbox ~/IdeaProjects/Sandbox
.idea
src
Hello.kt
Sandbox.iml
External Libraries
< 1.8 > /Library/Java/JavaVirtualMachines/jdk1
KotlinJavaRuntime
1:1
n/a
UTF-8

Kotlin: A new version 1.2.21-release-IJ2017.3-1 of the Kotlin plugin is available. Install (5 minutes ago)

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

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

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

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

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

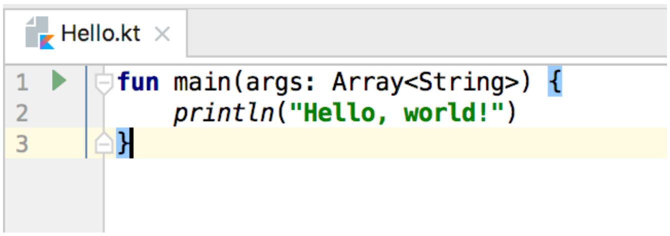
This particular function – the **`main`** function – has a special meaning in Kotlin. The **`main`** function indicates the starting place for your program. This is called the *application entry point*, and one such entry point must be defined for Sandbox (or any program) to be runnable. Every project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

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

Running your Kotlin file

Shortly after you finish typing the code in Listing 1.1, IntelliJ will display a green ▶, known as the "run button," to the left of the first line (Figure 1.11). (If the icon does not appear, or if you see a red line underneath the filename in the tab or under any of the code you entered, this means you have an error in your code. Double-check that you typed the code exactly as shown in Listing 1.1. On the other hand, if you see a red and blue Kotlin K, this flag is the same as the run button.)

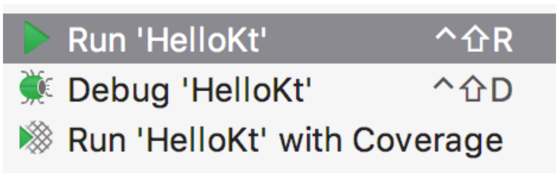
Figure 1.11 Run button



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

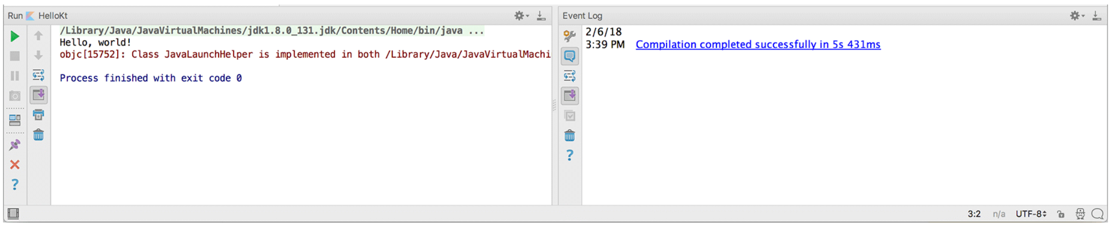
It is time for your program to come to life and greet the world. Click the run button. Select Run 'HelloKt' from the menu that appears (Figure 1.12). This tells IntelliJ you want to see your program in action.

Figure 1.12 Running `Hello.kt`



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

Figure 1.13 Run and event log tool windows



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java ...
Hello, world!
objc[15752]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachi

Process finished with exit code 0
```

```
2/6/18
3:39 PM   Compilation completed successfully in 5s 431ms
```

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

(macOS users, you may see red error text stating that there is an issue with `JavaLauncherHelper`, as shown in Figure 1.13. Do not worry about this. It is an unfortunate side effect of how the Java Runtime Environment is installed on macOS. To remove it would require a lot of effort, but the issue does no harm – so you may ignore it and carry on.)

On the right is the *event log tool window*, which displays information about work IntelliJ did to get your program ready to run. We will not mention the event log again, because you get much more interesting output in the console. (For the same reason, do not be concerned if the event log never opened to begin with.) You can close it with the hide button at its top right, which looks like this: ⤓.

Compilation and execution of Kotlin/JVM code

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

First, IntelliJ *compiles* the Kotlin code using the `kotlinc-jvm` compiler. This means IntelliJ translates the Kotlin code you wrote into *bytecode*, the language the JVM "speaks." If `kotlinc-jvm` has any problems translating your Kotlin code, it will display an error message (or messages) giving you a hint about how to fix the issues. Otherwise, if the compilation process goes smoothly, IntelliJ moves on to the execution phase.

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

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

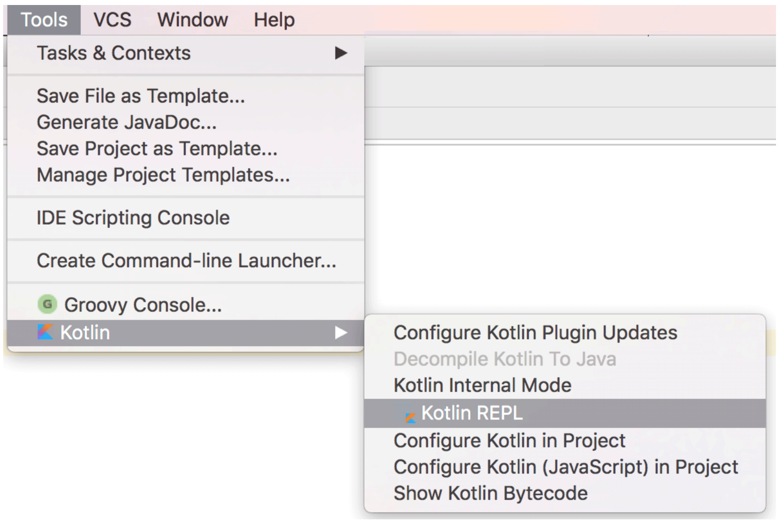
You will not need a comprehensive understanding of the Kotlin compilation process to work through this book. We will, however, discuss bytecode in more detail in Chapter 2.

The Kotlin REPL

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

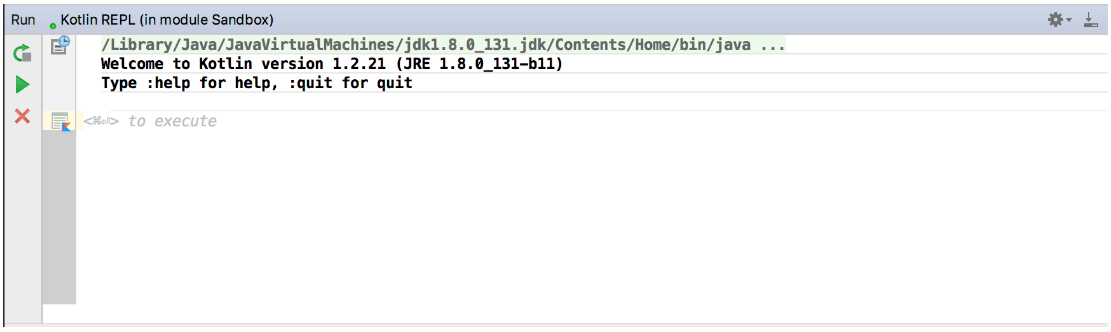
In IntelliJ, open the Kotlin REPL tool window by selecting Tools → Kotlin → Kotlin REPL (Figure 1.14).

Figure 1.14 Opening the Kotlin REPL tool window



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

Figure 1.15 The Kotlin REPL tool window



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java ...
Welcome to Kotlin version 1.2.21 (JRE 1.8.0_131-b11)
Type :help for help, :quit for quit
```

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

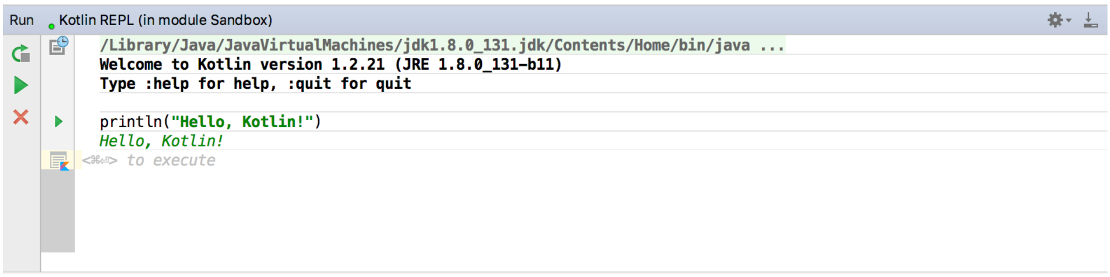
Enter the following code in the REPL:

Listing 1.2 “Hello, Kotlin!” (REPL)

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

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

Figure 1.16 Evaluating the code



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java ...
Welcome to Kotlin version 1.2.21 (JRE 1.8.0_131-b11)
Type :help for help, :quit for quit

println("Hello, Kotlin!")
Hello, Kotlin!
<⌘⏎> to execute
```

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

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

For the More Curious: Why Use IntelliJ?

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

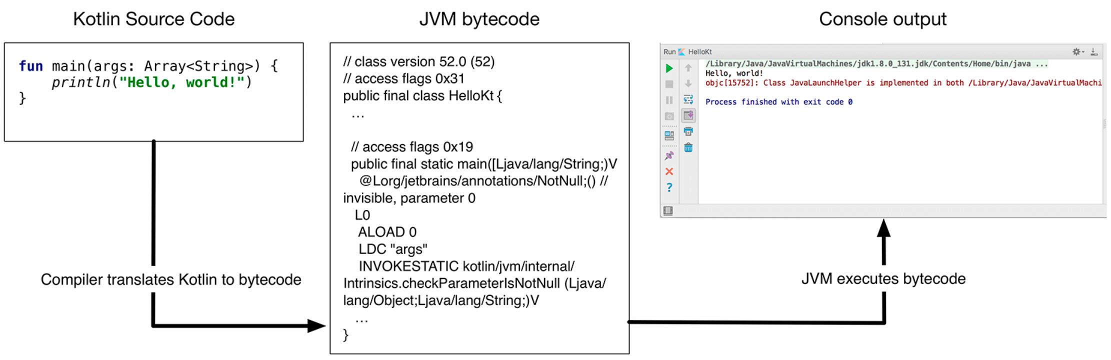
- write syntactically and semantically correct code with features like syntax highlighting, context-sensitive suggestions, and automatic code completion
- run and debug your code with features like debug breakpoints and real-time code stepping when your application is running
- restructure existing code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

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

For the More Curious: Targeting the JVM

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

Figure 1.17 Compilation and execution flow



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

```
// class version 52.0 (52)
// access flags 0x31
public final class HelloKt {
  ...

  // access flags 0x19
  public final static main([Ljava/lang/String;)V
    @Lorg/jetbrains/annotations/NotNull;() //
invisible, parameter 0
   L0
    ALOAD 0
    LDC "args"
    INVOKESTATIC kotlin/jvm/internal/
Intrinsics.checkParameterIsNotNull (Ljava/
lang/Object;Ljava/lang/String;)V
   ...
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java ...
Hello, world!
objc[15752]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachi

Process finished with exit code 0
```

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

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

Kotlin is not limited to the JVM, however. At the time of this writing, Kotlin can also be compiled into JavaScript or even into native binaries that run directly on a given platform – such as Windows, Linux, and macOS – negating the need for a virtual machine layer.

Challenge: REPL Arithmetic

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

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

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

Index

Symbols

!
  for platform types, 297
  logical 'not' operator, 33
`!!.` (double-bang/non-null assertion operator), 89
`!=` (non-equality operator), 29
`!==` (referential non-equality operator), 29
`$` (for string interpolation/templating), 40
`%` (modulus/remainder operator), 120
&& (logical 'and' operator), 33
+ (addition operator), 28, 234
++ (increment operator), 147
+= (addition and assignment operator), 15, 234
+= (plus assign operator), 135, 146, 154, 156
– (minus operator), 156
–= (minus assign operator), 135, 146, 156
–> (arrow operator)
  in anonymous function definitions, 71
  in function type definitions, 69
  in `when` expressions, 38
`.` (dot)
  for class property references, 168
  for function calls, 67
`..` (range operator), 37, 234
`:` operator
  for interface implementation, 244
  for subclassing, 207
`::` operator, for function references, 79
< (less-than operator), 29
<= (less-than-or-equal-to operator), 29
<> (for parameterized type definitions), 130
= (assignment operator)
  for maps, 156
  for variable values, 14
  in single-expression function syntax, 57
== (structural equality operator), 28, 234
=== (referential equality operator), 29
> (greater-than operator), 29, 234
>= (greater-than-or-equal-to operator), 29
`?.` (safe call operator), 88
`?:` (null coalescing operator), 91
`@JvmField` annotation, 307
`@JvmName` annotation, 303
`@JvmOverloads` annotation, 303
`@JvmStatic` annotation, 309
`@NotNull` annotation (Java), 101, 298
`@Nullable` annotation (Java), 297
`@Throws` annotation, 312
`[]` (get/index operator), 106, 130, 155, 234
`[]=` (set operator), 134
`\` (escape character), 104
_, for temporary variables, 188
`||` (logical 'or' operator), 33

A

abstract classes
  about, 243, 248-250
  abstract functions, 248
  interface implementation, 250
  subclassing, 249
  vs interfaces, 250
abstract functions, 248
`abstract` keyword, 248
`add` function, 133, 135, 146
`addAll` function, 135, 146
addition and assignment operator (+=), 15, 234
addition operator (+), 28, 234
algebraic data types, 240
`also` function, 126, 128, 185
`and(number)` function, 121
Android
  accessing view elements, 330, 333
  activities, 320
  Android KTX Kotlin extensions library, 341
  Anko Kotlin enhancements library, 342
  click listeners, 335
  creating a project, 318
  Gradle build automation tool, 322
  importing classes, 331
  Kotlin library dependencies, 323
  Kotlin plug-ins, 322
  Kotlin single abstract method conversions, 335
  Kotlin synthetic properties, 333
  lifecycle functions, 330
  manifest, 346
  project organization, 324
  saved instance state, 337-339
  SDK packages, 316
  threads, 348
  user interfaces, 324-327
  using coroutines, 349-351

(see also coroutines)
view element ids, 326
widgets, 326
Android Studio
about, 315-321
application logs, 348
emulators, 327
anonymous functions
about, 67-73
and function references, 78
and Kotlin standard library functions, 67, 123
arguments, 71
as closures, 80, 81
as function arguments, 75
calling, 68
defining, 68
implicit returns, 70
inlining, 77
`it` keyword, 72
lambda terminology, 75
memory use, 77
parameters, 71
vs Java anonymous inner classes, 82
`Any` class
about, 214
and platform independence, 217
default function implementations, 229
application entry point, 7
`apply` function, 123, 128
arguments
(see also functions)
in class constructors, default, 191
in class constructors, named, 192
in functions, default, 56
in functions, named, 59
to anonymous functions, 71
vs parameters, 51
`Array` type, 150
`ArrayIndexOutOfBoundsException`, 131
arrow operator (–>)
in anonymous function definitions, 71
in function type definitions, 69
in `when` expressions, 38
`as` operator
for renaming imported extensions, classes, 281
for type casting, 215
`assert` function, 98
assignment operator (=)
for maps, 156
for variable values, 14
in single-expression function syntax, 57
`async` function, 349
`await` function, 351

B

bitwise operations, 121
`Boolean` type, 16
`break` expression, 148
`by` keyword, 199
`Byte` type, 113
bytecode
about, 12
decompiling, 24
inspecting, 22

C

`Char` type, 16, 111
`checkNotNull` function, 97
`class` keyword, 241
classes
about, 163-172
abstract, 243, 248
(see also abstract classes)
`Any`, 214
(see also **`Any`** class)
class bodies, 165
class functions, 164
companion objects, 225
constructors, 164, 188
(see also constructors)
data classes, 228-231
(see also data classes)
declared with `object`, 219-225
(see also companion objects, object declarations, object expressions)
defining, 163
enumerated classes (enums), 231, 236
instantiating, 164
Kotlin type hierarchy, 214
nested, 226
object declarations, 220, 221
object expressions, 225
properties (see properties)
renaming imported classes, 281
scope, 178

sealed classes, 241
subclasses, 206
superclasses, 206
clear function, 135, 146
closures, 80, 81
code comments
about, 32
IntelliJ IDEA shortcut, 86
collection types
(see also `List` type, `Map` type, `Sequence` type, `Set` type)
compared, 162
creating instances, 129
eager vs lazy, 290
`companion` modifier, 225
companion objects, 225
compareTo function, 234
comparison operators
about, 29
order of operator precedence, 33
compilation, 9
compile-time constants, 21
compile-time errors, 86
compiler, 86
computed properties, 172
conditional expressions
about, 34
omitting curly braces in, 36
console, 8
`const` modifier, 21
constants, 13
constructors
about, 188-191
calling, 188, 190
default values for parameters, 189, 191
defining class properties in, 189
defining initialization logic in, 190
for subclassing, 207
named arguments, 192
parameters, 188
primary, 188, 189
secondary, 189
consumer role, 265
contains function, 132, 143, 234
containsAll function, 132, 143
control flow, 27
copy function, 230
coroutines
about, 349-351
async function, 349
await function, 351
coroutine builder functions, 351
enabling, 349
launch function, 350
suspending functions, 351
count function, 67
covariance, contravariance, 266

D

data classes
about, 228-231
benefits and limitations, 231
destructuring, 230
implementation of library functions, 229
`Deferred` type, 349
delegates
about, 199
lazy, 199
destructuring
about, 107, 142
using **split**, 107
distinct function, 149
dot syntax
for class property references, 168
for function calls, 67
`Double` type
about, 16, 113, 116
converting from `Int`, 117
converting to `Int`, 120
double-bang/non-null assertion operator (`!!.`), 89

E

editor, 5
elementAt function, 143
encapsulation (see visibility)
enumerated classes (enums), 231, 236
enumerated types, 231
equality (see referential equality, structural equality)
equals function
in data classes, 230
overriding, 234, 238
error function, 98
errors
compile-time, 86

runtime, 86
escape character (\), 104
escape sequences, 104
event log tool window, 9
exceptions
about, 92
ArrayIndexOutOfBoundsException, 131
custom, 95
Exception type, 96
IllegalStateException, 94
KotlinNullPointerException, 89
throwing, 94
unchecked, 100
unhandled, 93
Exercism project, 357-362
exitProcess function, 252
extensions
about, 269-284
bytecode representation, 276
defining extension functions, 270
extension files, 279
extension packages, 279
extension properties, 274
generic extension functions, 273
in the Kotlin standard library, 282
on nullable types, 275
on superclasses, 271
renaming imported extensions, 281
visibility, 279
extracting functions using IntelliJ IDEA
command, 43-45

F

field keyword, 169
files
creating, 6
running, 8
filter function, 287
final keyword, 211
first function, 131
flatMap function, 287
Float type, 113
floating points, 118
fold function, 288
for loops, 137
forEach function, 138
forEachIndexed function, 138
format function, 118
fun keyword, 47
function types
about, 69
as return types, 80
type inference with, 74
vs Java anonymous inner classes, 82
functional programming
about, 285-291
Arrow.kt library, 292
categories of functions, 285
combines, 288
composable functions, 286
filters, 287
higher-order functions, 287
immutability of variables, 286
transforms, 285
functions
(see also function types)
about, 7, 43
add, 133, 135, 146
addAll, 135, 146
also, 126, 128, 185
alternative syntax, 57
and(number), 121
anonymous, 67-73
(see also anonymous functions)
apply, 123, 128
arguments, 51
assert, 98
async, 349
await, 351
backtick naming syntax, 63
body, 46, 49
calling, 51
chaining calls, 272
checkNotNull, 97
class functions, 164, 178
clear, 135, 146
combining functions, 288
compareTo, 234
composable, 286
contains, 132, 143, 234
containsAll, 132, 143
copy, 230
count, 67
default arguments, 56
distinct, 149

dot syntax, 67
elementAt, 143
equals, 230, 234, 238
error, 98
exitProcess, 252
file-level, 61
filter, 287
filter functions, 287
first, 131
flatMap, 287
fold, 288
forEach, 138
forEachIndexed, 138
format, 118
from Kotlin standard library, 123, 128
function references, 78, 125
generateSequence, 290
get, 234
getOrDefault, 155
getOrElse, 131, 155
getOrNull, 131
getOrPut, 156
getValue, 155
hashCode, 238
header, 46-48
higher-order functions, 81
implicitly called, 124
in Java bytecode, 61
indexOf, 104
inlining, 77
intArrayOf, 150
inv(), 121
isInitialized, 198
iterator functions, 290
last, 131
launch, 350
let, 88, 92, 124, 128
listOf, 129
main, 7, 13
map, 285
mapOf, 153
measureNanoTime, 292
measureTimeInMillis, 292
mutableListOf, 133
mutableMapOf, 153
mutableSetOf, 145
mutator functions, 135, 146
named, 67
named arguments, 59
naming conventions, 47
operator overloading, 233
overloading, 62
overriding, 169
parameters, 48, 51
plus, 234
plusAssign, 234
precondition functions, 97
predicate functions, 287
println, 7
put, 156
putAll, 156
rangeTo, 234
readText, 346
remove, 133, 146, 156
removeAll, 146
removeIf, 135
replace, 107
require, 98
requireNotNull, 98
return statement, 49, 57
return type, 48, 57
roundToInt, 120
run, 125, 128, 257
scope, 50
setOf, 143
shl(bitcount), 121
shr(bitcount), 121
single-expression, 57
split, 106
structure of, 46
substring, 105
takeIf, 127, 128
takeUnless, 127
test function naming, 63
Thread.sleep, 253
to, 154
toBigDecimal, 116
toBinaryString, 121
toDouble, 116
toDoubleOrNull, 116
toFloat, 116
toInt, 120
toIntOrNull, 116
toList, 134, 149
toLong, 116
toMutableList, 134, 149

- **toMutableSet**, 149
- **toSet**, 149
- **toString**, 217, 229
- transform functions, 285
- transformer functions, 285
- `Unit` functions, 58
- **valueOf**, 236
- visibility, 47
- **with**, 126, 128
- **xor(number)**, 121
- **zip**, 288

G

generateSequence function, 290
generic type parameters, 255, 258
generics
- about, 255-268
- defining a generic function, 256
- defining a generic type, 255
- producers and consumers, 265
- type constraints, 259
- type erasure, 267
- type inference with, 256
- with `vararg`, 260

get function, 234
get/index operator (`[]`), 106, 130, 155, 234
getOrDefault function, 155
getOrElse function, 131, 155
getOrNull function, 131
getOrPut function, 156
getValue function, 155
greater-than operator (>), 29, 234
greater-than-or-equal-to operator (>=), 29

H

hashCode function, overriding, 238

I

`if/else` statements
- comparison operators in, 29
- defining, 27-37
- `else if` branches, 30, 38
- logical operators in, 32
- nested, 31
- omitting curly braces in, 36
- order of conditions, 30

`IllegalStateException`, 94
immutable, problems with terminology, 151, 184
imperative programming, 289
implicit returns, 70
`in` keyword
- for collections, 234
- for generic consumers, 265
- for ranges, 37
- in `for` loops, 138

increment operator (++), 147
indexes (see indices)
indexOf function, 104
indices
- about, 104
- **forEachIndexed** function, 138
- in lists, 130
- in strings, 104
- out of bounds, 131

`infix` keyword, 275
inheritance
- about, 205-211
- adding functionality in subclasses, 209
- creating subclasses, 206
- Kotlin type hierarchy, 214
- overriding superclass functionality in subclasses, 207

`init` keyword, 193
initialization
- about, 187-201
- delegates, 199
- initialization order, 196, 200, 201
- initializer blocks, 193, 200
- late, 198
- lazy, 198
- properties, 194-199

`inline` keyword, 77
`Int` type
- about, 14, 16, 113, 114
- converting from `Double`, 120
- converting to `Double`, 117

`IntArray` type, 150
intArrayOf function, 150
IntelliJ IDEA
- about, 1-7
- benefits, 11
- commenting code, 86
- console, 8
- displaying function return types, 58
- editor, 5

- error indicator, 15
- event log tool window, 9
- extracting functions, 43-45
- Kotlin bytecode tool window, 23
- opening a project, 13
- overriding **equals** and **hashCode**, 238
- project tool window, 5
- refactoring code, 43-45
- run tool window, 8
- running a project, 8
- Search Everywhere dialog, 22
- shortcut for adding **main** function, 13

interface keyword, 243
interfaces
- about, 243-248
- default property getters and functions, 248
- defining, 243
- implementing, 244
- vs abstract classes, 250

interoperating with Java
- about, 295-314
- @JvmField annotation, 307
- @JvmName annotation, 303
- @JvmOverloads annotation, 303
- @JvmStatic annotation, 309
- @NotNull annotation, 298
- @Nullable annotation, 297
- @Throws annotation, 312
- Android platform, 332
- arrays, 150
- backtick function naming syntax, 63
- exceptions, 310
- Java classes, 296
- Java fields and properties, 301, 332
- Java method overloading, 303
- Kotlin anonymous functions, 313
- Kotlin companion objects, 307
- Kotlin default parameters, 303
- Kotlin file-level functions, 303
- Kotlin function types, 313
- Kotlin functions defined on companion objects, 309
- null safety, 101, 297, 298
- platform types, 297
- type mapping, 299
- using function literals vs anonymous inner classes, 335

IntRange type
- about, 105
- as an Iterable, 139

inv() function, 121
is operator, 212
isInitialized function, 198
it keyword
- with **also**, 185
- with anonymous functions, 72
- with **forEach**, 156
- with **let**, 89, 124

Iterable types, 139
iteration
- about, 137-139
- break expression, 148
- with for, 137
- with **forEach**, 138
- with **forEachIndexed**, 138
- with while, 146

J

Java
- (see also interoperating with Java)
- @NotNull annotation, 101, 298
- @Nullable annotation, 297
- anonymous inner classes, 82
- arrays, 150
- benefits of Kotlin, xii
- checked exceptions, 100
- class-level variables, 300
- decompiled Kotlin bytecode, 24
- exceptions, 311
- getters and setters, 300
- null safety, 101
- nullable types, 84
- NullPointerException, 84
- package private visibility, 185
- primitive types, 25, 299
- referential and structural equality, 110

Java Development Kit (JDK), 4
Java Virtual Machine (JVM), targeting, 3, 12

K

Kotlin bytecode tool window, 23
Kotlin language, history, xii
Kotlin REPL, 10, 11
KotlinNullPointerException, 89

L

lambda, lambda expression, lambda result (see anonymous functions)
last function, 131
lateinit keyword, 198
launch function, 350
less-than operator (<), 29
less-than-or-equal-to operator (<=), 29
let function, 88, 92, 124, 128
List type
  (see also lists)
  about, 16, 129-142, 162
  as an Iterable, 139
  possibility of changing contents, 151
  size property, 287
  vs MutableList, 133
listOf function, 129
lists
  (see also List type, MutableList type)
  about, 129-142, 162
  accessing elements by index, 130
  checking for elements, 132
  converting to sequences, 291
  converting to/from sets, 149
  creating, 129
  mutable, 133
  read-only, 129
  var vs val, 133
logical operators
  about, 33
  order of operator precedence, 33
logical 'and' operator (&&), 33
logical 'not' operator (!), 33
logical 'or' operator (||), 33
Long type, 113

M

main function
  as application entry point, 7
  IntelliJ shortcut, 13
map function, 285
Map type
  (see also maps)
  about, 16, 153-162
  as an Iterable, 139
  size property, 287
  vs MutableMap, 156
mapOf function, 153
maps
  (see also Map type, MutableMap type)
  about, 153-162
  accessing values by key, 155
  adding entries, 156
  creating, 153
measureNanoTime function, 292
measureTimeInMillis function, 292
minus assign operator (-=), 135, 146, 156
minus operator (-), 156
modules, 5, 185
modulus/remainder operator (%), 120
MutableList type
  (see also lists)
  about, 133
  mutator functions, 135
  size property, 287
  vs List, 133
mutableListOf function, 133
MutableMap type
  (see also maps)
  mutator functions, 156
  size property, 287
  vs Map, 156
mutableMapOf function, 153
MutableSet type
  (see also sets)
  mutator functions, 146
  size property, 287
  vs Set, 146
mutableSetOf function, 145
mutator functions
  for lists, 135
  for maps, 156
  for sets, 146

N

non-equality operator (!=), 29
Nothing type, 60
null coalescing operator (?:), 91
null safety
  about, 83
  assert precondition function, 98
  checking values with != null, 90
  checkNotNull precondition function, 97

- double-bang (non-null assertion) operator (`!!.`), 89
- **`error`** precondition function, 98
- `IllegalStateException`, 94
- `KotlinNullPointerException`, 89
- non-null types, 84
- null coalescing operator (`?:`), 91
- null safety using conditionals, 90
- nullable types, 85
- **`require`** precondition function, 98
- **`requireNotNull`** precondition function, 98
- safe call operator (`?.`), 88
- `try/catch` statements, 96
- with **`let`**, 124

numeric types
- (see also individual types)
- about, 113-121
- bitwise operations, 121
- compared, 113
- converting from strings, 116
- for decimal values, 116
- for integers, 114
- maximum and minimum values, 113
- precision, 118

O

object declarations
- about, 220, 221
- calling, 221
- initializing, 220

object expressions, 225
`object` keyword, 219, 241
object-oriented programming
- about, 163
- benefits, 180
- encapsulation, 166
- inheritance, 205
- polymorphism, 210

`open` keyword, 206
`operator` modifier, 233
operator overloading, 233, 238
`out` keyword, 261, 265
`override` keyword, 207, 246

P

packages, 181
`Pair` type, 154
parameterized types, 130
parameters
- (see also functions)
- about, 48
- `it` keyword, 72
- of anonymous functions, 71
- vs arguments, 51

platform independence, 12, 217
plus assign operator (+=), 135, 146, 154, 156
`plus` function, 234
`plusAssign` function, 234
polymorphism, 210
precondition functions, 97
`println` function, 7
private visibility, 166
producer role, 265
project tool window, 5
projects
- about, 5
- creating, 2
- opening, 13
- organizing with modules, 185
- organizing with packages, 181
- running, 8

properties
- about, 167-172, 182-184
- accessibility to class functions, 178
- computed, 172
- defined with `val` or `var`, 167, 182
- defining in constructors, 189
- dot syntax, 168
- extension properties, 274
- fields, 168, 169, 172
- getters, 168
- initialization, 168, 194-199
- nullable, 184
- overriding getters and setters, 169
- race conditions, 185
- setters, 168, 169
- visibility, 171
- vs local variables, 196

protected visibility, 208
public visibility, 47
`put` function, 156
`putAll` function, 156

R

race conditions, 185
range operator (`..`), 37, 234
ranges, 37
`rangeTo` function, 234
`readText` function, 346
refactoring using IntelliJ IDEA command, 43-45
referential equality, 110
referential equality operator (===), 29
referential non-equality operator (`!`==), 29
`reified` keyword, 267
`remove` function, 133, 146, 156
`removeAll` function, 146
`removeIf` function, 135
REPL, 10, 11
`replace` function, 107
`require` function
  about, 98
  for parameter requirements, 98
`requireNotNull` function, 98
reserved keywords, 63
`return` keyword
  about, 49
  absent from anonymous functions, 70
  absent from `Unit` functions, 58
return type, 48, 57
`roundToInt` function, 120
`run` function, 125, 128, 257
run tool window, 8
runtime errors, 86

S

safe call operator (`?.`), 88
scope
  functions, 50
  relative scoping, 124
sealed classes, 241
`Sequence` type, 290
  (see also sequences)
sequences
  about, 290
  converting from lists, 291
  iterator functions, 290
set operator (`[]=`), 134
`Set` type
  (see also sets)
  about, 16, 142-146, 162
  as an `Iterable`, 139
  possibility of changing contents, 151
  `size` property, 287
  vs `MutableSet`, 146
`setOf` function, 143
sets
  (see also `Set` type, `MutableSet` type)
  about, 142-146, 162
  converting to/from lists, 149
  creating, 143
  index-based access, 143
  mutable, 146
`shl(bitcount)` function, 121
`Short` type, 113
`shr(bitcount)` function, 121
single-expression functions, 57
singletons, 219
smart casting, 90, 216
`split` function, 106
standard functions, 123, 128
static type checking, 15
string concatenation, 28
string interpolation/templating
  about, 40
  interpolating an expression, 40
`String` type
  (see also strings)
  about, 16, 103-112
strings
  about, 103-112
  accessing characters by index, 104
  converting to numeric types, 116
  extracting substrings, 103-107
  immutability, 109
structural equality, 110
structural equality operator (==), 28, 234
`substring` function, 105
`super` keyword, 209

T

`takeIf` function, 127, 128
`takeUnless` function, 127
`this` keyword
  in class constructors, 190
  in extension functions, 270
`Thread.sleep` function, 253
`throw` operator, 94, 95

to function, 154
toBigDecimal function, 116
toBinaryString function, 121
toDouble function, 116
toDoubleOrNull function, 116
toFloat function, 116
toInt function, 120
toIntOrNull function, 116
toList function, 134, 149
toLong function, 116
toMutableList function, 134, 149
toMutableSet function, 149
toSet function, 149
toString function, 217, 229
try/catch statements, 96
type casting, 215
type checking, 15
type inference
  about, 20
  with function types, 74
  with generics, 256
types
  (see also individual types)
  Array, 150
  Boolean, 16
  Byte, 113
  Char, 16, 111
  collection types, 129
  commonly used, 16
  Deferred, 349
  Double, 16, 113, 116
  Float, 113
  generic (see generics)
  Int, 14, 16, 113, 114
  IntArray, 150
  IntRange, 105
  Iterable, 139
  Kotlin vs Java, 25
  List, 16, 129, 162
  Long, 113
  Map, 16, 153
  MutableList, 133
  MutableMap, 156
  MutableSet, 146
  non-null, 84
  Nothing, 60
  nullable, 85
  numeric (see numeric types, individual type names)
  Pair, 154
  platform types, 297
  Sequence, 290
  Set, 16, 142, 162
  Short, 113
  String, 16, 103
  Unit, 58

U

Unicode characters, 111
Unit type, 58

V

val keyword, 17
valueOf function, 236
var keyword, 14
vararg keyword, 260
variables
  about, 13
  declaring, 14
  file-level, 50
  initialization requirements, 50
  local, 50
  read-only variables vs compile-time constants, 21
  temporary, 188
  val keyword, 17
  var keyword, 14
visibility
  about, 47, 166
  and encapsulation, 166
  class functions, 166
  default, 166
  internal, 166, 185
  modifiers, 47, 166
  private, 166
  properties, 171
  protected, 166, 208
  public, 166

W

when expressions
  about, 38, 39
  scoping conditions to arguments, 39
  vs if/else expressions, 38

`while` loops, 146
`with` function, 126, 128

X

`xor(number)` function, 121

Z

`zip` function, 288