O'REILLY®

Java 8 Lambdas

FUNCTIONAL PROGRAMMING FOR THE MASSES

Richard Warburton

O'REILLY®

Java 8 Lambdas

If you're a developer with core Java SE skills, this hands-on book takes you through the language changes in Java 8 triggered by the addition of lambda expressions. You'll learn through code examples, exercises, and fluid explanations how these anonymous functions will help you write simple, clean, library-level code that solves business problems.

Lambda expressions are a fairly simple change to Java, and the first part of the book shows you how to use them properly. Later chapters show you how lambda functions help you improve performance with parallelism, write simpler concurrent code, and model your domain more accurately, including building better DSLs.

- Use exercises in each chapter to help you master lambda expressions in Java 8 quickly
- Explore streams, advanced collections, and other Java 8 library improvements
- Leverage multicore CPUs and improve performance with data parallelism
- Use techniques to "lambdify" your existing codebase or library code
- Learn practical solutions for lambda expression unit testing and debugging
- Implement SOLID principles of object-oriented programming with lambdas
- Write concurrent applications that efficiently perform message passing and non-blocking I/O

"Crucially, this book gives you clear motivations as to why, where, and how you'd use lambdas in order to improve your code base."

—**Martijn Verburg**
CEO of jClarity and Java Champion

"I can't recommend this book highly enough, and it should be on the shelf of every Java developer looking to embrace the language enhancements in JDK 8."

—**Daniel Bryant**
CTO of Instant Access Technologies

Richard Warburton is an empirical technologist and solver of deep-dive technical problems. Recently he has been working on data analytics for high performance computing. As a leader in the London Java Community, he organizes the Adopt-a-JSR programs for Java 8 and the Openjdk Hackdays.

PROGRAMMING / JAVA

US $29.99 CAN $31.99

ISBN: 978-1-449-370770

Twitter: @oreillymedia
facebook.com/oreilly

Java 8 Lambdas

Richard Warburton

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Java 8 Lambdas

by Richard Warburton

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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meghan Blanchette
Production Editor: Melanie Yarbrough
Copyeditor: Nancy Kotary
Proofreader: Rachel Head
Indexer: WordCo Indexing Services
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

March 2014: First Edition

Revision History for the First Edition:

2014-03-13: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449370770* for release details.

ISBN: 978-1-449-37077-0

[LSI]

Table of Contents

Preface vii

1. Introduction 1
Why Did They Need to Change Java Again? 1
What Is Functional Programming? 2
Example Domain 3

2. Lambda Expressions 5
Your First Lambda Expression 5
How to Spot a Lambda in a Haystack 6
Using Values 8
Functional Interfaces 9
Type Inference 11
Key Points 13
Exercises 14

3. Streams 17
From External Iteration to Internal Iteration 17
What's Actually Going On 20
Common Stream Operations 21
collect(toList()) 22
map 22
filter 24
flatMap 25
max and min 26
A Common Pattern Appears 27
reduce 28
Putting Operations Together 30
Refactoring Legacy Code 31

Multiple Stream Calls 34
Higher-Order Functions 36
Good Use of Lambda Expressions 36
Key Points 37
Exercises 37
Advanced Exercises 39

4. Libraries. 41
Using Lambda Expressions in Code 41
Primitives 42
Overload Resolution 45
@FunctionalInterface 47
Binary Interface Compatibility 47
Default Methods 48
Default Methods and Subclassing 49
Multiple Inheritance 52
The Three Rules 53
Tradeoffs 54
Static Methods on Interfaces 54
Optional 55
Key Points 56
Exercises 57
Open Exercises 58

5. Advanced Collections and Collectors. 59
Method References 59
Element Ordering 60
Enter the Collector 62
Into Other Collections 62
To Values 63
Partitioning the Data 64
Grouping the Data 65
Strings 66
Composing Collectors 67
Refactoring and Custom Collectors 69
Reduction as a Collector 76
Collection Niceties 77
Key Points 78
Exercises 78

6. Data Parallelism. 81
Parallelism Versus Concurrency 81

Why Is Parallelism Important? 83
Parallel Stream Operations 83
Simulations 85
Caveats 88
Performance 89
Parallel Array Operations 92
Key Points 94
Exercises 94

7. Testing, Debugging, and Refactoring. 97
Lambda Refactoring Candidates 97
In, Out, In, Out, Shake It All About 98
The Lonely Override 98
Behavioral Write Everything Twice 99
Unit Testing Lambda Expressions 102
Using Lambda Expressions in Test Doubles 105
Lazy Evaluation Versus Debugging 106
Logging and Printing 106
The Solution: peek 107
Midstream Breakpoints 107
Key Points 108

8. Design and Architectural Principles. 109
Lambda-Enabled Design Patterns 110
Command Pattern 110
Strategy Pattern 114
Observer Pattern 117
Template Method Pattern 119
Lambda-Enabled Domain-Specific Languages 123
A DSL in Java 124
How We Got There 125
Evaluation 127
Lambda-Enabled SOLID Principles 127
The Single Responsibility Principle 128
The Open/Closed Principle 130
The Dependency Inversion Principle 134
Further Reading 137
Key Points 137

9. Lambda-Enabled Concurrency. 139
Why Use Nonblocking I/O? 139
Callbacks 140

Message Passing Architectures 144
The Pyramid of Doom 145
Futures 147
Completable Futures 149
Reactive Programming 152
When and Where 155
Key Points 155
Exercises 156

10. Moving Forward. 159

Index. 161

Preface

For years, functional programming has been considered the realm of a small band of specialists who consistently claimed superiority to the masses while being unable to spread the wisdom of their approach. The main reason I've written this book is to challenge both the idea that there's an innate superiority in the functional style and the belief that its approach should be relegated to a small band of specialists!

For the last two years in the London Java Community, I've been getting developers to try out Java 8 in some form or another. I've found that many of our members enjoy the new idioms and libraries that it makes available to them. They may reel at the terminology and elitism, but they love the benefits that a bit of simple functional programming provides to them. A common thread is how much easier it is to read code using the new Streams API to manipulate objects and collections, such as filtering out albums that were made in the UK from a `List` of all albums.

What I've learned when running these kinds of events is that examples matter. People learn by repeatedly digesting simple examples and developing an understanding of patterns out of them. I've also noticed that terminology can be very off-putting, so anytime there's a hard-sounding concept, I give an easy-to-read explanation.

For many people, what Java 8 offers by way of functional programming is incredibly limited: no monads,[1] no language-level lazy evaluation, no additional support for immutability. As pragmatic programmers, this is fine; what we want is the ability to write library-level abstractions so we can write simple, clean code that solves business problems. We're even happier if someone else has written these libraries for us and we can just focus on doing our daily jobs.

1. This is the only mention of this word in this book.

Why Should I Read This Book?

In this book we'll explore:

- How to write simpler, cleaner, and easier-to-read code—especially around collections
- How to easily use parallelism to improve performance
- How to model your domain more accurately and build better DSLs
- How to write less error-prone and simpler concurrent code
- How to test and debug your lambda expressions

Developer productivity isn't the only reason why lambda expressions have been added to Java; there are fundamental forces in our industry at work here as well.

Who Should Read This Book?

This book is aimed squarely at Java developers who already have core Java SE skills and want to get up to speed on the big changes in Java 8.

If you're interested in reading about lambda expressions and how they can improve your lot as a professional developer, read on! I don't assume you know about lambda expressions themselves, or any of the core library changes; instead, I introduce concepts, libraries, and techniques from scratch.

Although I would love for every developer who has ever lived to go and buy this book, realistically, it's not appropriate for everyone. If you don't know any Java at all, this isn't the book for you. At the same time, though lambda expressions in Java are very well covered here, I don't explain how they are used in any other languages.

I don't provide a basic introduction to the use of several facets of the Java SE, such as collections, anonymous inner classes, or the event handling mechanism in Swing. I assume that you already know about all of these elements.

How to Read This Book

This book is written in an example-driven style: very soon after a concept is introduced, you'll see some code. Occasionally you might see something in the code that you're not 100% familar with. Don't worry—it'll be explained very soon afterward, frequently in the next paragraph.

This approach also lets you try out the ideas as you go along. In fact, at the end of most chapters there are further examples for you to practice on your own. I highly recommend that you try doing these katas as you get to the end of the chapter. Practice makes perfect,

and—as every pragmatic programmer knows—it's really easy to fool yourself into thinking that you understand some code when in reality you've missed a detail.

Because the use of lambda expressions is all about abstracting complexity away into libraries, I introduce a bunch of common library niceties as I go along. Chapters 2 through 6 cover the core language changes and also the improved libraries that JDK 8 brings.

The final three chapters are about applying functional programming in the wild. I'll talk about a few tricks that make testing and debugging code a bit easier in Chapter 7. Chapter 8 explains how existing principles of good software design also apply to lambda expressions. Then I talk about concurrency and how to use lambda expressions to write concurrent code that's easy to understand and maintain in Chapter 9. These chapters also introduce third-party libraries, where relevant.

It's probably worth thinking of the opening four chapters as the introductory material —things that everyone will need to know to use Java 8 properly. The latter chapters are more complex, but they also teach you how to be a more complete programmer who can confidently use lambda expressions in your own designs. There are also exercises as you go along, and the answers to these can be found on GitHub. If you practice the exercises as you go along, you'll soon master lambda expressions.

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/RichardWarburton/java-8-lambdas-exercises*.

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 a CD-ROM of 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 require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Java 8 Lambdas* by Richard Warburton (O'Reilly). Copyright 2014 Richard Warburton, 978-1-449-37077-0."

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*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database

from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

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. You can access this page at *http://oreil.ly/java_8_lambdas*.

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

While the name on the cover of this book is mine, many other people have been influential and helpful in its publication.

Thanks should go firstly to my editor, Meghan, and the team at O'Reilly for making this process a pleasurable experience and accelerating their deadlines where appropriate. It was great to be introduced to Meghan by Martijn and Ben to begin with; this book would never have happened without that meeting.

The review process was a huge step in improving the overall quality of the book, and my heartfelt appreciation goes out to those who have helped as part of the formal and informal review process, including Martijn Verburg, Jim Gough, John Oliver, Edward

Wong, Brian Goetz, Daniel Bryant, Fred Rosenberger, Jaikiran Pai, and Mani Sarkar. Martijn in particular has been hugely helpful with his battle-won advice on writing a technical book.

It would also be remiss of me to ignore the Project Lambda development team at Oracle. Updating an established language is a big challenge, and they've done a great job in Java 8 of giving me something fun to write about and support. The London Java Community also deserves its share of praise for being so actively involved and supportive when helping to test out the early Java release and making it so easy to see what kinds of mistakes developers make and what can be fixed.

A lot of people have been incredibly supportive and helpful while I was going through the effort of writing a book. I'd like to specifically call out my parents, who have always been there whenever they were needed. It has also been great to have encouragement and positive comments from friends such as old compsoc members, especially Sadiq Jaffer and the Boys Brigade.

CHAPTER 1
Introduction

Before we begin our exploration of what lambda expressions are and how we can use them, you should at least understand why they exist to begin with. In this chapter, I'll cover that and also explain the structure and motivation of this book.

Why Did They Need to Change Java Again?

Java 1.0 was released in January 1996, and the world of programming has changed quite a bit since then. Businesses are requiring ever more complex applications, and most programs are executed on machines with powerful multicore CPUs. The rise of Java Virtual Machines (JVM), with efficient runtime compilers has meant that programmers can focus more on writing clean, maintainable code, rather than on code that's efficiently using every CPU clock cycle and every byte of memory.

The elephant in the room is the rise of multicore CPUs. Programming algorithms involving locks is error-prone and time-consuming. The `java.util.concurrent` package and the wealth of external libraries have developed a variety of concurrency abstractions that begin to help programmers write code that performs well on multicore CPUs. Unfortunately, we haven't gone far enough—until now.

There are limits to the level of abstractions that library writers can use in Java today. A good example of this is the lack of efficient parallel operations over large collections of data. Java 8 allows you to write complex collection-processing algorithms, and simply by changing a single method call you can efficiently execute this code on multicore CPUs. In order to enable writing of these kinds of bulk data parallel libraries, however, Java needed a new language change: lambda expressions.

Of course there's a cost, in that you must learn to write and read lambda-enabled code, but it's a good trade-off. It's easier for programmers to learn a small amount of new syntax and a few new idioms than to have to handwrite a large quantity of complex thread-safe code. Good libraries and frameworks have significantly reduced the cost

and time associated with developing enterprise business applications, and any barrier to developing easy-to-use and efficient libraries should be removed.

Abstraction is a concept that is familiar to us all from object-oriented programming. The difference is that object-oriented programming is mostly about abstracting over data, while functional programming is mostly about abstracting over behavior. The real world has both of these things, and so do our programs, so we can and should learn from both influences.

There are other benefits to this new abstraction as well. For many of us who aren't writing performance-critical code all the time, these are more important wins. You can write easier-to-read code—code that spends time expressing the intent of its business logic rather than the mechanics of how it's achieved. Easier-to-read code is also easier to maintain, more reliable, and less error-prone.

You don't need to deal with the verbosity and readbility issues surrounding anonymous inner classes when writing callbacks and event handlers. This approach allows programmers to work on event processing systems more easily. Being able to pass functions around easily also makes it easier to write lazy code that initializes values only when necessary.

In addition, the language changes that enable the additional collection methods, `default` methods, can be used by everyday programmers who are maintaining their own libraries.

It's not your grandfather's Java any longer, and that's a good thing.

What Is Functional Programming?

Functional programming is a term that means different things to different people. At the heart of functional programming is thinking about your problem domain in terms of immutable values and functions that translate between them.

The communities that have developed around different programming languages each tend to think that the set of features that have been incorporated into their language are the key ones. At this stage, it's a bit too early to tell how Java programmers will define functional programming. In a sense, it's unimportant; what we really care about is writing *good* code rather than functional code.

In this book, I focus on pragmatic functional programming, including techniques that can be used and understood by most developers and that help them write programs that are easier to read and maintain.

Example Domain

Throughout the book, examples are structured around a common problem domain: music. Specifically, the examples represent the kind of information you might see on albums. Here's a brief summary of the terms:

Artist
: An individual or group who creates music

  - *name*: The name of the artist (e.g., "The Beatles")
  - *members*: A set of other artists who comprise this group (e.g., "John Lennon"); this field might be empty
  - *origin*: The primary location of origin of the group (e.g., "Liverpool").

Track
: A single piece of music

  - *name*: The name of the track (e.g., "Yellow Submarine")

Album
: A single release of music, comprising several tracks

  - *name*: The name of the album (e.g., "Revolver")
  - *tracks*: A list of tracks
  - *musicians*: A list of artists who helped create the music on this album

This domain is used to illustrate how to use functional programming techniques within a normal business domain or Java application. You may not consider it the perfect example subject, but it's simple, and many of the code examples in this book will bear similarity to those that you may see in your business domain.

CHAPTER 2

Lambda Expressions

The biggest language change in Java 8 is the introduction of lambda expressions—a compact way of passing around behavior. They are also a pretty fundamental building block that the rest of this book depends upon, so let's get into what they're all about.

Your First Lambda Expression

Swing is a platform-agnostic Java library for writing graphical user interfaces (GUIs). It has a fairly common idiom in which, in order to find out what your user did, you register an *event listener*. The event listener can then perform some action in response to the user input (see Example 2-1).

Example 2-1. Using an anonymous inner class to associate behavior with a button click

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

In this example, we're creating a new object that provides an implementation of the `ActionListener` class. This interface has a single method, `actionPerformed`, which is called by the `button` instance when a user actually clicks the on-screen button. The anonymous inner class provides the implementation of this method. In Example 2-1, all it does is print out a message to say that the button has been clicked.

This is actually an example of using *code as data*—we're giving the button an object that represents an action.

Anonymous inner classes were designed to make it easier for Java programmers to pass around code as data. Unfortunately, they don't make it easy enough. There are still four lines of boilerplate code required in order to call the single line of important logic. Look how much gray we get if we color out the boilerplate:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

Boilerplate isn't the only issue, though: this code is fairly hard to read because it obscures the programmer's intent. We don't want to pass in an object; what we really want to do is pass in some behavior. In Java 8, we would write this code example as a lambda expression, as shown in Example 2-2.

Example 2-2. Using a lambda expression to associate behavior with a button click

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Instead of passing in an object that implements an interface, we're passing in a block of code—a function without a name. `event` is the name of a parameter, the same parameter as in the anonymous inner class example. `->` separates the parameter from the body of the lambda expression, which is just some code that is run when a user clicks our button.

Another difference between this example and the anonymous inner class is how we declare the variable `event`. Previously, we needed to explicitly provide its type—`ActionEvent event`. In this example, we haven't provided the type at all, yet this example still compiles. What is happening under the hood is that `javac` is inferring the type of the variable `event` from its context—here, from the signature of `addActionListener`. What this means is that you don't need to explicitly write out the type when it's obvious. We'll cover this inference in more detail soon, but first let's take a look at the different ways we can write lambda expressions.

Although lambda method parameters require less boilerplate code than was needed previously, they are still statically typed. For the sake of readability and familiarity, you have the option to include the type declarations, and sometimes the compiler just can't work it out!

How to Spot a Lambda in a Haystack

There are a number of variations of the basic format for writing lambda expressions, which are listed in Example 2-3.

Example 2-3. Some different ways of writing lambda expressions

```
Runnable noArguments = () -> System.out.println("Hello World");  ❶

ActionListener oneArgument = event -> System.out.println("button clicked");  ❷

Runnable multiStatement = () -> {  ❸
   System.out.print("Hello");
   System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;  ❹

BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  ❺
```

❶ shows how it's possible to have a lambda expression with no arguments at all. You can use an empty pair of parentheses, `()`, to signify that there are no arguments. This is a lambda expression implementing `Runnable`, whose only method, `run`, takes no arguments and is a `void` return type.

❷ we have only one argument to the lambda expression, which lets us leave out the parentheses around the arguments. This is actually the same form that we used in Example 2-2.

Instead of the body of the lambda expression being just an expression, in ❸ it's a full block of code, bookended by curly braces (`{}`). These code blocks follow the usual rules that you would expect from a method. For example, you can return or throw exceptions to exit them. It's also possible to use braces with a single-line lambda, for example to clarify where it begins and ends.

Lambda expressions can also be used to represent methods that take more than one argument, as in ❹. At this juncture, it's worth reflecting on how to *read* this lambda expression. This line of code doesn't add up two numbers; it creates a function that adds together two numbers. The variable called `add` that's a `BinaryOperator<Long>` isn't the result of adding up two numbers; it is code that adds together two numbers.

So far, all the types for lambda expression parameters have been inferred for us by the compiler. This is great, but it's sometimes good to have the option of explicitly writing the type, and when you do that you need to surround the arguments to the lambda expression with parentheses. The parentheses are also necessary if you've got multiple arguments. This approach is demonstrated in ❺.

The *target type* of a lambda expression is the type of the context in which the lambda expression appears—for example, a local variable that it's assigned to or a method parameter that it gets passed into.

What is implicit in all these examples is that a lambda expression's type is context dependent. It gets inferred by the compiler. This target typing isn't entirely new, either. As shown in Example 2-4, the types of array initializers in Java have always been inferred from their contexts. Another familiar example is `null`. You can know what the type of `null` is only once you actually assign it to something.

Example 2-4. The righthand side doesn't specify its type; it is inferred from the context

```
final String[] array = { "hello", "world" };
```

Using Values

When you've used anonymous inner classes in the past, you've probably encountered a situation in which you wanted to use a variable from the surrounding method. In order to do so, you had to make the variable `final`, as demonstrated in Example 2-5. Making a variable `final` means that you can't reassign to that variable. It also means that whenever you're using a `final` variable, you know you're using a specific value that has been assigned to the variable.

Example 2-5. A final local variable being captured by an anonymous inner class

```
final String name = getUserName();
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("hi " + name);
    }
});
```

This restriction is relaxed a bit in Java 8. It's possible to refer to variables that aren't `final`; however, they still have to be *effectively* `final`. Although you haven't declared the variable(s) as `final`, you still cannot use them as nonfinal variable(s) if they are to be used in lambda expressions. If you do use them as nonfinal variables, then the compiler will show an error.

The implication of being effectively `final` is that you can assign to the variable only once. Another way to understand this distinction is that lambda expressions capture *values*, not variables. In Example 2-6, `name` is an effectively `final` variable.

Example 2-6. An effectively final variable being captured by an anonymous inner class

```
String name = getUserName();
button.addActionListener(event -> System.out.println("hi " + name));
```

I often find it easier to read code like this when the `final` is left out, because it can be just line noise. Of course, there are situations where it can be easier to understand code with an explicit `final`. Whether to use the effectively `final` feature comes down to personal choice.

If you assign to the variable multiple times and then try to use it in a lambda expression, you'll get a compile error. For example, Example 2-7 will fail to compile with the error message: `local variables referenced from a lambda expression must be final or effectively final`.

Example 2-7. Fails to compile due to the use of a not effectively final variable

```
String name = getUserName();
name = formatUserName(name);
button.addActionListener(event -> System.out.println("hi " + name));
```

This behavior also helps explain one of the reasons some people refer to lambda expressions as "closures." The variables that aren't assigned to are *closed* over the surrounding state in order to bind them to a value. Among the chattering classes of the programming language world, there has been much debate over whether Java really has closures, because you can refer to only effectively `final` variables. To paraphrase Shakespeare: *A closure by any other name will function all the same*. In an effort to avoid such pointless debate, I'll be referring to them as "lambda expressions" throughout this book. Regardless of what we call them, I've already mentioned that lambda expressions are statically typed, so let's investigate the types of lambda expressions themselves: these types are called *functional interfaces*.

Functional Interfaces

A functional interface is an interface with a single abstract method that is used as the type of a lambda expression.

In Java, all method parameters have types; if we were passing `3` as an argument to a method, the parameter would be an `int`. So what's the type of a lambda expression?

There is a really old idiom of using an interface with a single method to represent a method and reusing it. It's something we're all familiar with from programming in Swing, and it is exactly what was going on in Example 2-2. There's no need for new magic to be employed here. The exact same idiom is used for lambda expressions, and we call this kind of interface a *functional interface*. Example 2-8 shows the functional interface from the previous example.

Example 2-8. The ActionListener interface: from an ActionEvent to nothing

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent event);
}
```

`ActionListener` has only one abstract method, `actionPerformed`, and we use it to *represent* an action that takes one argument and produces no result. Remember, because `actionPerformed` is defined in an interface, it doesn't actually need the `abstract` keyword in order to be abstract. It also has a parent interface, `EventListener`, with no methods at all.

So it's a functional interface. It doesn't matter what the single method on the interface is called—it'll get matched up to your lambda expression as long as it has a compatible method signature. Functional interfaces also let us give a useful name to the type of the parameter—something that can help us understand what it's used for and aid readability.

The functional interface here takes one `ActionEvent` parameter and doesn't return anything (`void`), but functional interfaces can come in many kinds. For example, they may take two parameters and return a value. They can also use generics; it just depends upon what you want to use them for.

From now on, I'll use diagrams to represent the different kinds of functional interfaces you're encountering. The arrows going into the function represent arguments, and if there's an arrow coming out, it represents the return type. For example, an `ActionListener` would look like Figure 2-1.

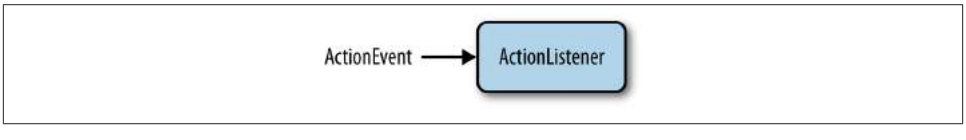

Figure 2-1. The ActionListener interface showing an ActionEvent going in and nothing (void) coming out

Over time you'll encounter many functional interfaces, but there is a core group in the Java Development Kit (JDK) that you will see time and time again. I've listed some of the most important functional interfaces in Table 2-1.

Table 2-1. Important functional interfaces in Java

Interface name	Arguments	Returns	Example
`Predicate<T>`	`T`	`boolean`	Has this album been released yet?
`Consumer<T>`	`T`	`void`	Printing out a value
`Function<T,R>`	`T`	`R`	Get the name from an `Artist` object
`Supplier<T>`	None	`T`	A factory method
`UnaryOperator<T>`	`T`	`T`	Logical not (`!`)
`BinaryOperator<T>`	`(T, T)`	`T`	Multiplying two numbers (`*`)

I've talked about what types functional interfaces take and mentioned that `javac` can automatically infer the types of parameters and that you can manually provide them, but when do you know whether to provide them? Let's look a bit more at the details of type inference.

Type Inference

There are certain circumstances in which you need to manually provide type hints, and my advice is to do what you and your team find easiest to read. Sometimes leaving out the types removes line noise and makes it easier to see what is going on. Sometimes leaving them in can make it clearer what is going on. I've found that at first they can sometimes be helpful, but over time you'll switch to adding them in only when they are actually needed. You can figure out whether they are needed from a few simple rules that I'll introduce in this chapter.

The type inference used in lambdas is actually an extension of the target type inference introduced in Java 7. You might be familiar with Java 7 allowing you to use a `diamond` operator that asks `javac` to *infer* the generic arguments for you. You can see this in Example 2-9.

Example 2-9. Diamond inference for variables

```
Map<String, Integer> oldWordCounts = new HashMap<String, Integer>();  ❶
Map<String, Integer> diamondWordCounts = new HashMap<>();  ❷
```

For the variable `oldWordCounts` ❶ we have explicitly added the generic types, but `diamondWordCounts` ❷ uses the `diamond` operator. The generic types aren't written out—the compiler just figures out what you want to do by itself. Magic!

It's not really magic, of course. Here, the generic types to `HashMap` can be inferred from the type of `diamondWordCounts` ❷. You still need to provide generic types on the variable that is being assigned to, though.

If you're passing the constructor straight into a method, it's also possible to infer the generic types from that method. In Example 2-10, we pass a `HashMap` as an argument that already has the generic types on it.

Example 2-10. Diamond inference for methods

```
useHashmap(new HashMap<>());

...

private void useHashmap(Map<String, String> values);
```

In the same way that Java 7 allowed you to leave out the generic types for a constructor, Java 8 allows you to leave out the types for whole parameters of lambda expressions.

Again, it's not magic: `javac` looks for information close to your lambda expression and uses this information to figure out what the correct type should be. It's still type checked and provides all the safety that you're used to, but you don't have to state the types explicitly. This is what we mean by *type inference*.

It's also worth noting that in Java 8 the type inference has been improved. The earlier example of passing `new HashMap<>()` into a `use Hashmap` method actually wouldn't have compiled in Java 7, even though the compiler had all the information it needed to figure things out.

Let's go into a little more detail on this point with some examples.

In both of these cases we're assigning the variables to a functional interface, so it's easier to see what's going on. The first example (Example 2-11) is a lambda that tells you whether an `Integer` is greater than 5. This is actually a `Predicate`—a functional interface that checks whether something is true or false.

Example 2-11. Type inference

```
Predicate<Integer> atLeast5 = x -> x > 5;
```

A `Predicate` is also a lambda expression that returns a value, unlike the previous `ActionListener` examples. In this case we've used an expression, `x > 5`, as the body of the lambda expression. When that happens, the return value of the lambda expression is the value its body evaluates to.

You can see from Example 2-12 that `Predicate` has a single generic type; here we've used an `Integer`. The only argument of the lambda expression implementing `Predicate` is therefore inferred as an `Integer`. `javac` can also check whether the return value is a `boolean`, as that is the return type of the `Predicate` method (see Figure 2-2).

Example 2-12. The predicate interface in code, generating a boolean from an Object

```
public interface Predicate<T> {
    boolean test(T t);
}
```

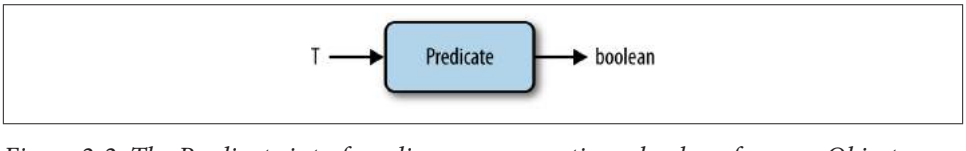

Figure 2-2. The Predicate interface diagram, generating a boolean from an Object

Let's look at another, slightly more complex functional interface example: the `BinaryOperator` interface, which is shown in Example 2-13. This interface takes two arguments and returns a value, all of which are the same type. In the code example we've used, this type is `Long`.

Example 2-13. A more complex type inference example

```
BinaryOperator<Long> addLongs = (x, y) -> x + y;
```

The inference is smart, but if it doesn't have enough information, it won't be able to make the right decision. In these cases, instead of making a wild guess it'll just stop what it's doing and ask for help in the form of a compile error. For example, if we remove some of the type information from the previous example, we get the code in Example 2-14.

Example 2-14. Code doesn't compile due to missing generics

```
BinaryOperator add = (x, y) -> x + y;
```

This code results in the following error message:

```
Operator '&#x002B;' cannot be applied to java.lang.Object, java.lang.Object.
```

That looks messy: what is going on here? Remember that `BinaryOperator` was a functional interface that had a generic argument. The argument is used as the type of both arguments, `x` and `y`, and also for its return type. In our code example, we didn't give any generics to our `add` variable. It's the very definition of a *raw* type. Consequently, our compiler thinks that its arguments and return values are all instances of `java.lang.Object`.

We will return to the topic of type inference and its interaction with method overloading in "Overload Resolution" on page 45, but there's no need to understand more detail until then.

Key Points

- A lambda expression is a method without a name that is used to pass around behavior as if it were data.
- Lambda expressions look like this: `BinaryOperator<Integer> add = (x, y) → x + y`.
- A functional interface is an interface with a single abstract method that is used as the type of a lambda expression.

Exercises

At the end of each chapter is a series of exercises to give you an opportunity to practice what you've learned during the chapter and help you learn the new concepts. The answers to these exercises can be found on GitHub.

1. Questions about the `Function` functional interface (Example 2-15).

   Example 2-15. The function functional interface

   ```
   public interface Function<T, R> {
       R apply(T t);
   }
   ```

   a. Can you draw this functional interface diagrammatically?

   b. What kind of lambda expressions might you use this functional interface for if you were writing a software calculator?

   c. Which of these lambda expressions are valid `Function<Long,Long>` implementations?

   ```
   x -> x + 1;
   (x, y) -> x + 1;
   x -> x == 1;
   ```

2. *ThreadLocal lambda expressions*. Java has a class called `ThreadLocal` that acts as a container for a value that's local to your current thread. In Java 8 there is a new factory method for `ThreadLocal` that takes a lambda expression, letting you create a new `ThreadLocal` without the syntactic burden of subclassing.

   a. Find the method in Javadoc or using your IDE.

   b. The Java `DateFormatter` class isn't thread-safe. Use the constructor to create a thread-safe `DateFormatter` instance that prints dates like this: "01-Jan-1970".

3. *Type inference rules*. Here are a few examples of passing lambda expressions into functions. Can `javac` infer correct argument types for the lambda expressions? In other words, will they compile?

   a. `Runnable helloWorld = () -> System.out.println("hello world");`

   b. The lambda expression being used as an `ActionListener`:

   ```
   JButton button = new JButton();
   button.addActionListener(event ->
     System.out.println(event.getActionCommand()));
   ```

   c. Would `check(x -> x > 5)` be inferred, given the following overloads for `check`?

   ```
   interface IntPred {
       boolean test(Integer value);
   }
   ```

```
boolean check(Predicate<Integer> predicate);

boolean check(IntPred predicate);
```

You might want to look up the method argument types in Javadoc or in your IDE in order to determine whether there are multiple valid overloads.

CHAPTER 3

Streams

The language changes introduced in Java 8 are intended to help us write better code. New core libraries are a key part of that, so in this chapter we start to look at them. The most important core library changes are focused around the Collections API and its new addition: *streams*. Streams allow us to write collections-processing code at a higher level of abstraction.

The `Stream` interface contains a series of functions that we'll explore throughout this chapter, each of which corresponds to a common operation that you might perform on a `Collection`.

From External Iteration to Internal Iteration

A lot of the examples in this chapter and the rest of the book refer to domain classes, which were introduced in "Example Domain" on page 3.

A common pattern for Java developers when working with collections is to iterate over a collection, operating on each element in turn. For example, if we wanted to add up the number of musicians who are from London, we would write the code in Example 3-1.

Example 3-1. Counting London-based artists using a for loop

```
int count = 0;
for (Artist artist : allArtists) {
    if (artist.isFrom("London")) {
        count++;
    }
}
```

There are several problems with this approach, though. It involves a lot of boilerplate code that needs to be written every time you want to iterate over the collection. It's also hard to write a parallel version of this `for` loop. You would need to rewrite every `for` loop individually in order to make them operate in parallel.

Finally, the code here doesn't fluently convey the intent of the programmer. The boilerplate `for` loop structure obscures meaning; to understand anything we must read though the body of the loop. For a single `for` loop, doing this isn't too bad, but when you have a large code base full of them it becomes a burden (especially with nested loops).

Looking under the covers a little bit, the `for` loop is actually syntactic sugar that wraps up the iteration and hides it. It's worth taking a moment to look at what's going on under the hood here. The first step in this process is a call to the `iterator` method, which creates a new `Iterator` object in order to control the iteration process. We call this *external iteration*. The iteration then proceeds by explicitly calling the `hasNext` and `next` methods on this `Iterator`. Example 3-2 demonstrates the expanded code in full, and Figure 3-1 shows the pattern of method calls that happen.

Example 3-2. Counting London-based artists using an iterator

```
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

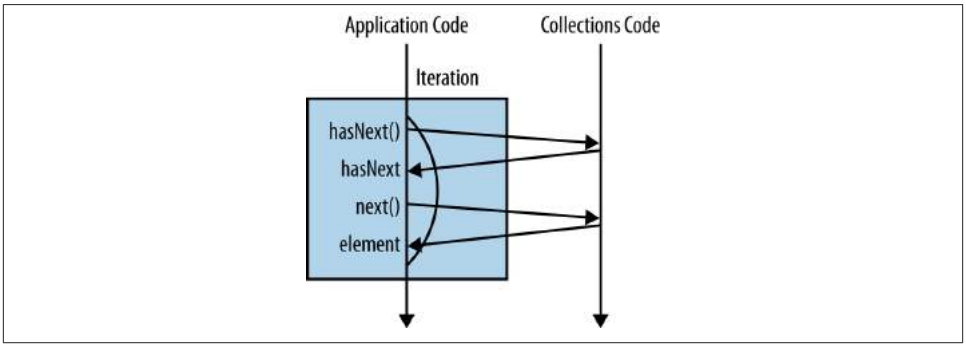

Figure 3-1. External iteration

External iteration has some negative issues associated with it, too. First, it becomes hard to abstract away the different behavioral operations that we'll encounter later in this

chapter. It is also an approach that is inherently serial in nature. The big-picture issue here is that using a `for` loop conflates what you are doing with how you are doing it.

An alternative approach, *internal iteration*, is shown in Example 3-3. The first thing to notice is the call to `stream()`, which performs a similar role to the call to `iterator()` in the previous example. Instead of returning an `Iterator` to control the iteration, it returns the equivalent interface in the internal iteration world: `Stream`.

Example 3-3. Counting London-based artists using internal iteration

```
long count = allArtists.stream()
                       .filter(artist -> artist.isFrom("London"))
                       .count();
```

Figure 3-2 depicts the flow of method calls with respect to the library; compare it with Figure 3-1.

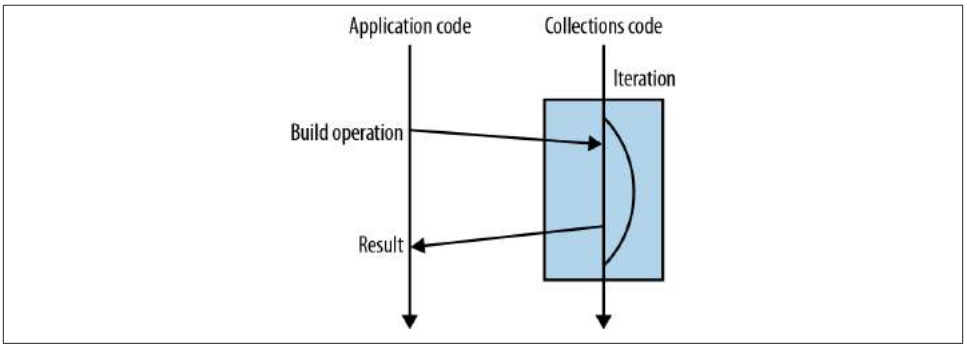

Figure 3-2. Internal iteration

A `Stream` is a tool for building up complex operations on collections using a functional approach.

We can actually break this example into two simpler operations:

- Finding all the artists from London
- Counting a list of artists

Both of these operations correspond to a method on the `Stream` interface. In order to find artists from London, we `filter` the `Stream`. Filtering in this case means "keep only objects that pass a test." The test is defined by a function, which returns either `true` or `false` depending on whether the artist is from London. Because we're practicing func-

tional programming when using the Streams API, we aren't changing the contents of the `Collection`; we're just declaring what the contents of the `Stream` will be. The `count()` method counts how many objects are in a given `Stream`.

What's Actually Going On

When I wrote the previous example, I broke it up into two simpler operations: filtering and counting. You may think that this is pretty wasteful—when I wrote the `for` loop in Example 3-1, there was only one loop. It looks like you would need two here, as there are two operations. In fact, the library has been cleverly designed so that it iterates over the list of artists only once.

In Java, when you call a method it traditionally corresponds to the computer actually doing something; for example, `System.out.println("Hello World");` prints output to your terminal. Some of the methods on `Stream` work a little bit differently. They are normal Java methods, but the `Stream` object returned isn't a new collection—it's a recipe for creating a new collection. So just think for a second about what the code in Example 3-4 does. Don't worry if you get stuck—I'll explain in a bit!

Example 3-4. Just the filter, no collect step

```
allArtists.stream()
          .filter(artist -> artist.isFrom("London"));
```

It actually doesn't do very much at all—the call to `filter` builds up a `Stream` recipe, but there's nothing to force this recipe to be used. Methods such as `filter` that build up the `Stream` recipe but don't force a new value to be generated at the end are referred to as *lazy*. Methods such as `count` that generate a final value out of the `Stream` sequence are called *eager*.

The easiest way of seeing that is if we add in a `println` statement as part of the filter in order to print out the artists' names. Example 3-5 is a modified version of Example 3-4 with such a printout. If we run this code, the program doesn't print anything when it's executed.

Example 3-5. Not printing out artist names due to lazy evaluation

```
allArtists.stream()
          .filter(artist -> {
              System.out.println(artist.getName());
              return artist.isFrom("London");
          });
```

If we add the same printout to a stream that has a terminal step, such as the counting operation from Example 3-3, then we will see the names of our artists printed out (Example 3-6).

Example 3-6. Printing out artist names

```
long count = allArtists.stream()
                       .filter(artist -> {
                           System.out.println(artist.getName());
                           return artist.isFrom("London");
                       })
                       .count();
```

So, if you ran Example 3-6 with the members of The Beatles as your list of artists, then you would see Example 3-7 printed out on your command line.

Example 3-7. Sample output showing the members of The Beatles being printed

```
John Lennon
Paul McCartney
George Harrison
Ringo Starr
```

It's very easy to figure out whether an operation is eager or lazy: look at what it returns. If it gives you back a `Stream`, it's lazy; if it gives you back another value or `void`, then it's eager. This makes sense because the preferred way of using these methods is to form a sequence of lazy operations chained together and then to have a single eager operation at the end that generates your result. This is how our counting example operates, but it's the simplest case: only two operations.

This whole approach is somewhat similar to the familiar *builder* pattern. In the builder pattern, there are a sequence of calls that set up properties or configuration, followed by a single call to a `build` method. The object being created isn't created until the call to `build` occurs.

I'm sure you're asking, "Why would we want to have the differentiator between lazy and eager options?" By waiting until we know more about what result and operations are needed, we can perform the computations more efficiently. A good example is finding the first number that is `> 10`. We don't need to evaluate all the elements to figure this out—only enough to find our first match. It also means that we can string together lots of different operations over our collection and iterate over the collection only once.

Common Stream Operations

At this point, it's worth just having a look back at some common `Stream` operations in order to get more of a feel of what's available in the API. As we will cover only a few important examples, I recommend looking at the Javadoc for the new API to see what else is available.

collect(toList())

`collect(toList())` is an eager operation that generates a list from the values in a `Stream`.

The values in the `Stream` that are operated on are derived from the initial values and the recipe produced by the sequence of `Stream` calls. In fact, `collect` is a very general and powerful construct, and we'll look into its other uses in more detail in Chapter 5. Here's an example of this operation:

```
List<String> collected = Stream.of("a", "b", "c")  ❶
                               .collect(Collectors.toList());  ❷

assertEquals(Arrays.asList("a", "b", "c"), collected);  ❸
```

This example shows how `collect(toList())` can be used to build a result list out of a `Stream`. It's important to remember, as discussed in the previous section, that because many `Stream` functions are lazy, you do need to use an eager operation such as `collect` at the end of a sequence of chained method calls.

This example also shows the general format for all the examples in this section. It starts by taking a `Stream` from a `List` ❶. There is some operation, followed by collecting into a list ❷. Finally, we perform an assert to show you what the results are equal to ❸.

You can think of the opening call to `stream` and the closing call to a `collect` or other terminal method as *bun* methods. They aren't the actual filling of our stream burger, but they do help us see where the operations begin and end.

map

If you've got a function that converts a value of one type into another, `map` lets you apply this function to a stream of values, producing another stream of the new values.

You'll probably notice fairly soon that you've been doing some kind of map operations for years already. Say you are writing Java code that takes a list of strings and converts them to their uppercase equivalents. You would loop over all the values in the list and call `toUppercase` on each element. You would then add each of the resulting values into a new `List`. Example 3-8 is code written in this style.

Example 3-8. Converting strings to uppercase equivalents using a for loop

```
List<String> collected = new ArrayList<>();
for (String string : asList("a", "b", "hello")) {
    String uppercaseString = string.toUpperCase();
    collected.add(uppercaseString);
}

assertEquals(asList("A", "B", "HELLO"), collected);
```

`map` is one of the most commonly used `Stream` operations (see Figure 3-3). You could probably have guessed this, given how frequently you have implemented something similar to the aforementioned `for` loop. Example 3-9 is the same example of turning a list of strings into their uppercase equivalents using the stream framework.

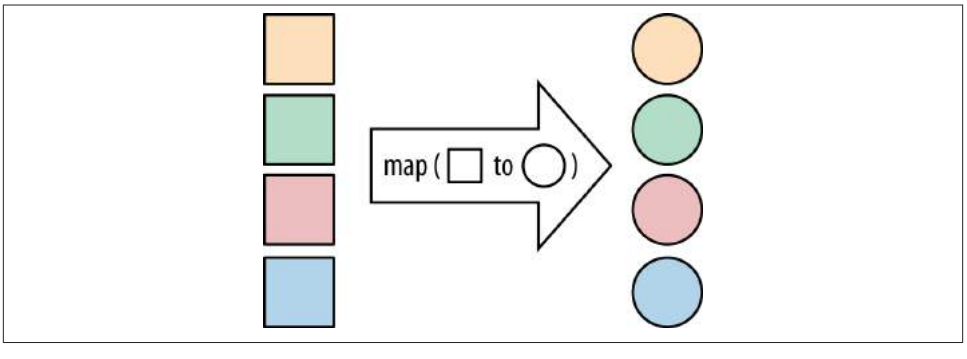

Figure 3-3. The map operation

Example 3-9. Converting strings to uppercase equivalents using map

```
List<String> collected = Stream.of("a", "b", "hello")
                               .map(string -> string.toUpperCase())  ❶
                               .collect(toList());

assertEquals(asList("A", "B", "HELLO"), collected);
```

The lambda expression passed into `map` ❶ both takes a `String` as its only argument and returns a `String`. It isn't necessary for both the argument and the result to be the same type, but the lambda expression passed in must be an instance of `Function` (Figure 3-4). This is a generic functional interface with only one argument.

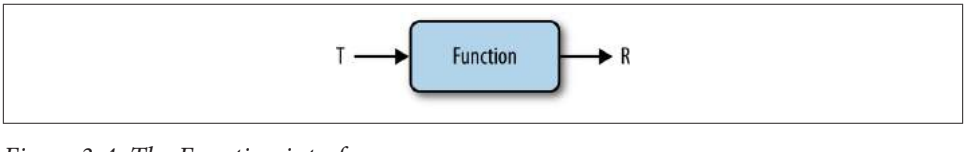

Figure 3-4. The Function interface

filter

Any time you're looping over some data and checking each element, you might want to think about using the new `filter` method on `Stream` (see Figure 3-5).

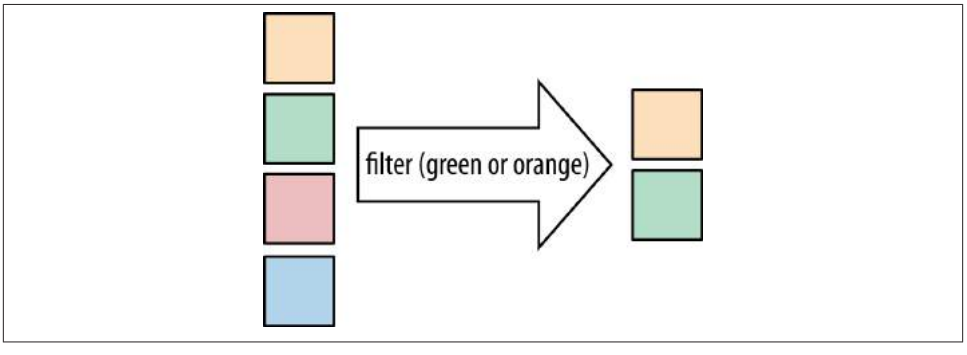

Figure 3-5. The filter operation

We've already looked at a `filter` example, so you may want to skip this section if you feel familiar with the concept. Still here? Good! Suppose we've got a list of strings and we want to find all the strings that start with a digit. So, `"1abc"` would be accepted and `"abc"` wouldn't. We might write some code that loops over a list and uses an `if` statement to see what the first character is, something like the code in Example 3-10.

Example 3-10. Looping over a list and using an if statement

```
List<String> beginningWithNumbers = new ArrayList<>();
for(String value : asList("a", "1abc", "abc1")) {
    if (isDigit(value.charAt(0))) {
        beginningWithNumbers.add(value);
    }
}

assertEquals(asList("1abc"), beginningWithNumbers);
```

I'm sure you've written some code that looks like this: it's called the `filter` pattern. The central idea of `filter` is to retain some elements of the `Stream`, while throwing others out. Example 3-11 shows how you would write the same code in a functional style.

Example 3-11. Functional style

```
List<String> beginningWithNumbers
  = Stream.of("a", "1abc", "abc1")
```

```
            .filter(value -> isDigit(value.charAt(0)))
            .collect(toList());

assertEquals(asList("1abc"), beginningWithNumbers);
```

Much like `map`, `filter` is a method that takes just a single function as an argument—here we're using a lambda expression. This function does the same job that the expression in the `if` statement did earlier. Here, it returns `true` if the `String` starts with a digit. If you're refactoring legacy code, the presence of an `if` statement in the middle of a `for` loop is a pretty strong indicator that you really want to use `filter`.

Because this function is doing the same job as the `if` statement, it must return either `true` or `false` for a given value. The `Stream` after the filter has the elements of the `Stream` beforehand, which evaluated to `true`. The functional interface for this type of function is our old friend from the previous chapter, the `Predicate` (shown in Figure 3-6).

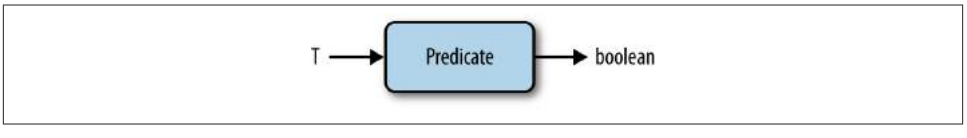

Figure 3-6. The Predicate interface

flatMap

`flatMap` (see Figure 3-7) lets you replace a value with a `Stream` and concatenates all the streams together.

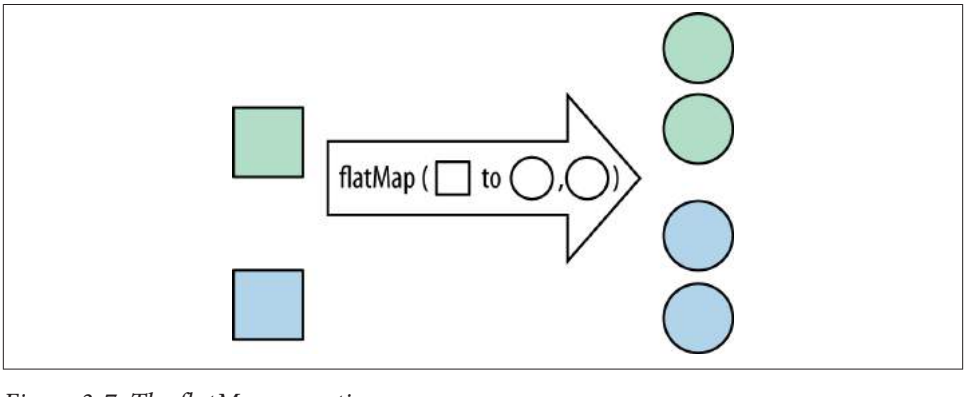

Figure 3-7. The flatMap operation

You've already seen the `map` operation, which replaces a value in a `Stream` with a new value. Sometimes you want a variant of `map` in which you produce a new `Stream` object as the replacement. Frequently you don't want to end up with a stream of streams, though, and this is where `flatMap` comes in handy.

Let's look at a simple example. We've got a `Stream` of lists of numbers, and we want all the numbers from these in sequences. We can solve this problem using an approach like the one in Example 3-12.

Example 3-12. Stream list

```
List<Integer> together = Stream.of(asList(1, 2), asList(3, 4))
                               .flatMap(numbers -> numbers.stream())
                               .collect(toList());

assertEquals(asList(1, 2, 3, 4), together);
```

In each case, we replace the `List` with a `Stream` using the `stream` method, and `flatMap` does the rest. Its associated functional interface is the same as `map`'s—the `Function`—but its return type is restricted to streams and not any value.

max and min

A pretty common operation that we might want to perform on streams is finding the maximum or minimum element. Fortunately, this case is very well covered by the `max` and `min` operations that are provided by the Streams API. As a demonstration of these operations, Example 3-13 provides some code that finds the shortest track on an album. In order to make it easier to see that we've got the right result, I've explicitly listed the tracks on this album in the code snippet; I'll admit that it's not the best-known album.

Example 3-13. Finding the shortest track with streams

```
List<Track> tracks = asList(new Track("Bakai", 524),
                            new Track("Violets for Your Furs", 378),
                            new Track("Time Was", 451));

Track shortestTrack = tracks.stream()
                            .min(Comparator.comparing(track -> track.getLength()))
                            .get();

assertEquals(tracks.get(1), shortestTrack);
```

When we think about maximum and minimum elements, the first thing we need to think about is the ordering that we're going to be using. When it comes to finding the shortest track, the ordering is provided by the length of the tracks.

In order to inform the `Stream` that we're using the length of the track, we give it a `Comparator`. Conveniently, Java 8 has added a static method called `comparing` that lets

us build a comparator using keys. Previously, we always encountered an ugly pattern in which we had to write code that got a field out of both the objects being compared, then compare these field values. Now, to get the same element out of both elements being compared, we just provide a getter function for the value. In this case we'll use `length`, which is a getter function in disguise.

It's worth reflecting on the `comparing` method for a moment. This is actually a function that takes a function and returns a function. Pretty meta, I know, but also incredibly useful. At any point in the past, this method could have been added to the Java standard library, but the poor readability and verbosity issues surrounding anonymous inner classes would have made it impractical. Now, with lambda expressions, it's convenient and concise.

It's now possible for `max` to be called on an empty `Stream` so that it returns what's known as an `Optional` value. An `Optional` value is a bit like an alien: it represents a value that may exist, or may not. If our `Stream` is empty, then it won't exist; if it's not empty, then it will. Let's not worry about the details of `Optional` for the moment, since we'll be discussing it in detail in "Optional" on page 55. The only thing to remember is that we can pull out the value by calling the `get` method.

A Common Pattern Appears

`max` and `min` are both forms of a more general pattern of coding. The easiest way to see this is by taking our code from Example 3-13 and rewriting it into a `for` loop: we'll then extract the general pattern. Example 3-14 performs the same role as Example 3-13: it finds the shortest track on an album, but using a `for` loop.

Example 3-14. Finding the shortest track with a for loop

```
List<Track> tracks = asList(new Track("Bakai", 524),
                            new Track("Violets for Your Furs", 378),
                            new Track("Time Was", 451));

Track shortestTrack = tracks.get(0);
for (Track track : tracks) {
    if (track.getLength() < shortestTrack.getLength()) {
        shortestTrack = track;
    }
}

assertEquals(tracks.get(1), shortestTrack);
```

The code starts by initializing our `shortestTrack` variable with the first element of the list. Then it goes through the tracks. If there's a shorter track, it replaces the `shortestTrack`. At the end, our `shortestTrack` variable contains its namesake. Doubtless you've written thousands of `for` loops in your coding career, and many of them follow this pattern. The pseudocode in Example 3-15 characterizes the general form.

Example 3-15. The reduce pattern

```
Object accumulator = initialValue;
for(Object element : collection) {
    accumulator = combine(accumulator, element);
}
```

An `accumulator` gets pushed through the body of the loop, with the final value of the `accumulator` being the value that we were trying to compute. The `accumulator` starts with an `initialValue` and then gets folded together with each `element` of the list by calling `combine`.

The things that differ between implementations of this pattern are the `initialValue` and the `combine` function. In the original example, we used the first element in the list as our `initialValue`, but it doesn't have to be. In order to find the shortest value, our `combine` returned the shorter track of out of the current `element` and the `accumulator`.

We'll now take a look at how this general pattern can be codified by an operation in the Streams API itself.

reduce

Use the `reduce` operation when you've got a collection of values and you want to generate a single result. In earlier examples, we used the `count`, `min`, and `max` methods, which are all in the standard library because they are common use cases. All of these are forms of reduction.

Let's demonstrate the `reduce` operation by adding up streams of numbers. The overall pattern is demonstrated in Figure 3-8. We start with a count of 0—the count of an empty `Stream`—and fold together each element with an accumulator, adding the element to the accumulator at every step. When we reach the final `Stream` element, our accumulator has the sum of all the elements.

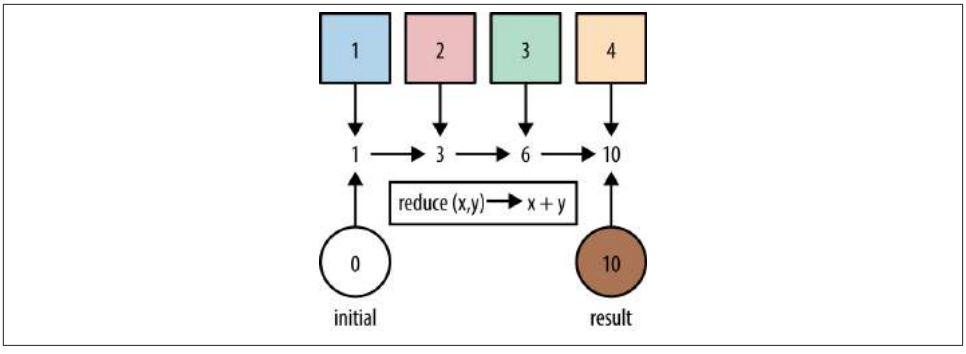

Figure 3-8. Implementing addition using the reduce operation

Example 3-16 shows what is going on in code. The lambda expression, known as a *reducer*, performs the summing and takes two arguments. `acc` is the accumulator and holds the current sum. It is also passed in the current element in the `Stream`.

Example 3-16. Implementing sum using reduce

```
int count = Stream.of(1, 2, 3)
                  .reduce(0, (acc, element) -> acc + element);

assertEquals(6, count);
```

The lambda expression returns the new `acc` value, which is the previous `acc` added to the current element. The type of the reducer is a `BinaryOperator`, which we encountered in Chapter 2.

"Primitives" on page 42 also refers to an implementation of `sum` within the standard library, which is recommended instead of the approach shown in this example in real code.

Table 3-1 shows the intermediate values for these variables for each element in the `Stream`. In fact, we could expand all the function applications that `reduce` to produce the code in Example 3-17.

Example 3-17. Expanding the application of reduce

```
BinaryOperator<Integer> accumulator = (acc, element) -> acc + element;
int count = accumulator.apply(
                accumulator.apply(
                    accumulator.apply(0, 1),
                2),
            3);
```

Table 3-1. Evaluating a sum reduce

element	acc	Result
N/A	N/A	0
1	0	1
2	1	3
3	3	6

Let's look at the equivalent imperative Java code, written in Example 3-18, so we can see how the functional and imperative versions match up.

Example 3-18. Imperative implementation of summing

```
int acc = 0;
for (Integer element : asList(1, 2, 3)) {
    acc = acc + element;
}
assertEquals(6, acc);
```

In the imperative version, we can see that the accumulator is a variable we update on every loop iteration. We also update it by adding the element. The loop is external to the collection and all updates to the variable are managed manually.

Putting Operations Together

With so many different operations related to the `Stream` interface, it can sometimes seem like you're wandering around a labyrinth looking for what you want. So let's work through a problem and see how it breaks down into simple `Stream` operations.

Our first problem to solve is, for a given album, to find the nationality of every band playing on that album. The artists who play each track can be solo artists or they can be in a band. We're going to use domain knowledge and artistic license to pretend that a band is really an artist whose name begins with *The*. This isn't exactly right, but it's pretty close!

The first thing to recognize is that the solution isn't just the simple application of any individual API call. It's not transforming the values like a `map`, it's not filtering, and it's not just getting a single value out of a `Stream` at the end. We can break the problem down into parts:

1. Get all the artists for an album.
2. Figure out which artists are bands.
3. Find the nationalities of each band.
4. Put together a set of these values.

Now it's easier to see how these steps fit into the API:

1. There's a nice `getMusicians` method on our `Album` class that returns a `Stream`.
2. We use `filter` to trim down the artists to include only bands.
3. We use `map` to turn the band into its nationality.
4. We use `collect(toList())` to put together a list of these nationalities.

When we put everything together, it ends up like this:

```
Set<String> origins = album.getMusicians()
                           .filter(artist -> artist.getName().startsWith("The"))
                           .map(artist -> artist.getNationality())
                           .collect(toSet());
```

This example shows the idiom of chaining operations a bit more clearly. The calls to `musicians`, `filter`, and `map` all return `Stream` objects, so they are lazy, while the `collect` method is eager. The `map` method is another function that takes just a lambda and whose purpose is to apply the function to every element in the `Stream`, returning a new `Stream`.

Our domain class here is actually quite convenient for us, in that it returns a `Stream` when we want to get a list of the musicians on our album. In your existing domain classes, you probably don't have a method that returns streams—you return existing collection classes such as `List` or `Set`. This is OK; all you need to do is call the `stream` method on your `List` or `Set`.

Now is probably a good time to think about whether you really want to expose `List` and `Set` objects in your domain model, though. Perhaps a `Stream` factory would be a better choice. The big win of only exposing collections via `Stream` is that it better encapsulates your domain model's data structure. It's impossible for any use of your domain classes to affect the inner workings of your `List` or `Set` simply by exposing a `Stream`.

It also encourages users of your domain class to write code in a more modern Java 8 style. It's possible to incrementally refactor to this style by keeping your existing getters and adding new `Stream`-returning getters. Over time, you can rewrite your legacy code until you've finally deleted all getters that return a `List` or `Set`. This kind of refactoring feels really good once you've cleared out all the legacy code!

Refactoring Legacy Code

Having talked a bit about refactoring already, let's look at an example of some legacy collections code that uses loops to perform a task and iteratively refactor it into a stream-

based implementation. At each step of the refactor, the code continues to pass its tests, though you'll either have to trust me on that one or test it yourself!

This example finds the names of all tracks that are over a minute in length, given some albums. Our legacy code is shown in Example 3-19. We start off by initializing a `Set` that we'll store all the track names in. The code then iterates, using a `for` loop, over all the albums, then iterates again over all the tracks in an album. Once we've found a track, we check whether the length is over 60 seconds, and if it is the name gets added to a `Set` of names.

Example 3-19. Legacy code finding names of tracks over a minute in length

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    for(Album album : albums) {
        for (Track track : album.getTrackList()) {
            if (track.getLength() > 60) {
                String name = track.getName();
                trackNames.add(name);
            }
        }
    }
    return trackNames;
}
```

We've stumbled across this code in our code base and noticed that it has a couple of nested loops. It's not quite clear what the purpose of this code is just from looking at it, so we decide to undertake our refactor. (There are lots of different approaches to refactoring legacy code for using streams—this is just one. In fact, once you are more familiar with the API itself, it's pretty likely that you won't need to proceed in such small steps. It serves educational purposes here to go a bit slower than you would in your professional job.)

The first thing that we're going to change is the `for` loops. We'll keep their bodies in the existing Java coding style for now and move to using the `forEach` method on `Stream`. This can be a pretty handy trick for intermediate refactoring steps. Let's use the `stream` method on our album list in order to get the first stream. It's also good to remember from the previous section that our domain already has the `getTracks` method on the album, which provides us a `Stream` of tracks. The code after we've completed step 1 is listed in Example 3-20.

Example 3-20. Refactor step 1: finding names of tracks over a minute in length

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
          .forEach(album -> {
              album.getTracks()
                   .forEach(track -> {
```

```
                    if (track.getLength() > 60) {
                        String name = track.getName();
                        trackNames.add(name);
                    }
                });
        });
    return trackNames;
}
```

In step 1, we moved to using streams, but we didn't really get their full potential. In fact, if anything the code is even less pretty than it was to begin with—d'oh! So, it's high time we introduced a bit more stream style into our coding. The inner `forEach` call looks like a prime target for refinement.

We're really doing three things here: finding only tracks over a minute in length, getting their names, and adding their names into our name `Set`. That means we need to call three `Stream` operations in order to get the job done. Finding tracks that meet a criterion sounds like a job for `filter`. Transforming tracks into their names is a good use of `map`. For the moment we're still going to add the tracks to our `Set`, so our terminal operation will still be a `forEach`. If we split out the inner `forEach` block, we end up with Example 3-21.

Example 3-21. Refactor step 2: finding names of tracks over a minute in length

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
          .forEach(album -> {
              album.getTracks()
                   .filter(track -> track.getLength() > 60)
                   .map(track -> track.getName())
                   .forEach(name -> trackNames.add(name));
          });
    return trackNames;
}
```

Now we've replaced our inner loop with something a bit more streamy, but we still have this pyramid of doom in our code. We don't really want to have nested stream operations; we want one simple and clean sequence of method calls.

What we really want to do is find a way of transforming our album into a stream of tracks. We know that whenever we want to *transform* or *replace* code, the operation to use is `map`. This is the more complex case of `map`, `flatMap`, for which the output value is also a `Stream` and we want them merged together. So, if we replace that `forEach` block with a `flatMap` call, we end up at Example 3-22.

Example 3-22. Refactor step 3: finding names of tracks over a minute in length

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();

    albums.stream()
          .flatMap(album -> album.getTracks())
          .filter(track -> track.getLength() > 60)
          .map(track -> track.getName())
          .forEach(name -> trackNames.add(name));

    return trackNames;
}
```

That looks a lot better, doesn't it? Instead of two nested `for` loops, we've got a single clean sequence of method calls performing the entire operation. It's not quite there yet, though. We're still creating a `Set` by hand and adding every element in at the end. We really want the entire computation to just be a chain of `Stream` calls.

I haven't yet shown you the recipe for this transformation, but you've met one of its friends. Just as you can use `collect(toList())` to build up a `List` of values at the end, you can also use `collect(toSet())` to build up a `Set` of values. So, we replace our final `forEach` call with this `collect` call, and we can now delete the `trackNames` variable, arriving at Example 3-23.

Example 3-23. Refactor step 4: finding names of tracks over a minute in length

```
public Set<String> findLongTracks(List<Album> albums) {
    return albums.stream()
                 .flatMap(album -> album.getTracks())
                 .filter(track -> track.getLength() > 60)
                 .map(track -> track.getName())
                 .collect(toSet());
}
```

In summary, we've taken a snippet of legacy code and refactored it to use idiomatic streams. At first we just converted to introduce streams and didn't introduce any of the useful operations on streams. At each subsequent step, we moved to a more idiomatic coding style. One thing that I haven't mentioned thus far but that was very helpful when actually writing the code samples is that at each step of the way I continued to run unit tests in order to make sure the code worked. Doing so is very helpful when refactoring legacy code.

Multiple Stream Calls

Rather than chaining the method calls, you could force the evaluation of each function individually following a sequence of steps. *Please* don't do this. Example 3-24 shows our

earlier origins of bands example written in that style. The original example is shown in Example 3-25 in order to make the comparison easier.

Example 3-24. Stream misuse

```
List<Artist> musicians = album.getMusicians()
                              .collect(toList());

List<Artist> bands = musicians.stream()
                              .filter(artist -> artist.getName().startsWith("The"))
                              .collect(toList());

Set<String> origins = bands.stream()
                           .map(artist -> artist.getNationality())
                           .collect(toSet());
```

Example 3-25. Idiomatically chained stream calls

```
Set<String> origins = album.getMusicians()
                           .filter(artist -> artist.getName().startsWith("The"))
                           .map(artist -> artist.getNationality())
                           .collect(toSet());
```

There are several reasons why the version in Example 3-24 is worse than the idiomatic, chained version:

- It's harder to read what's going on because the ratio of boilerplate code to actual business logic is worse.
- It's less efficient because it requires eagerly creating new collection objects at each intermediate step.
- It clutters your method with meaningless garbage variables that are needed only as intermediate results.
- It makes operations harder to automatically parallelize.

Of course, if you're writing your first few `Stream`-based examples, it's perfectly normal to write code that's a little bit like this. But if you find yourself writing blocks of operations like this often, you should stand back and see whether you can refactor them into a more concise and readable form.

If at this stage you feel uncomfortable with the amount of method chaining in the API, that's entirely natural. With more experience and more time these concepts will begin to feel quite natural, and it's not a reason to write Java code that splits up chains of operations as in Example 3-24. Ensuring that you format the code line by line, as you would when using the builder pattern, will boost your comfort level as well.

Higher-Order Functions

What we've repeatedly encountered throughout this chapter are what functional programmers call *higher-order functions*. A higher-order function is a function that either takes another function as an argument or returns a function as its result. It's very easy to spot a higher-order function: just look at its signature. If a functional interface is used as a parameter or return type, you have a higher-order function.

`map` is a higher-order function because its `mapper` argument is a function. In fact, nearly all the functions that we've encountered on the `Stream` interface are higher-order functions. In our earlier sorting example, we also used the `comparing` function. `comparing` not only took another function in order to extract an index value, but also returns a new `Comparator`. You might think of a `Comparator` as an object, but it has only a single abstract method, so it's a functional interface.

In fact, we can make a stronger statement than that. `Comparator` was invented when a function was needed, but all Java had at the time was objects, so we made a type of class —an anonymous class—that we could treat like a function. Being an object was always accidental. Functional interfaces are a step in the direction that we actually want.

Good Use of Lambda Expressions

When I first introduced lambda expressions, I gave the example of a callback that printed something out. That's a perfectly valid lambda expression, but it's not really helping us write simpler and more abstract code because it's still telling the computer to perform an operation. Removing the boilerplate was nice, but it's not the only improvement we get with lambda expressions in Java 8.

The concepts introduced in this chapter let us write simpler code, in the sense that they describe operations on data by saying *what* transformation is made rather than *how* the transformation occurs. This gives us code that has less potential for bugs and expresses the programmer's intent directly.

Another aspect of getting to the *what* and not the *how* is the idea of a *side effect–free* function. These are important because we can understand the full implications of what the functions are doing just by looking at what values they return.

Functions with no side effects don't change the state of anything else in the program or the outside world. The first lambda expression in this book had side effects because it printed some output on the console—an *observable* side effect of the function. What about the following example?

```
private ActionEvent lastEvent;

private void registerHandler() {
    button.addActionListener((ActionEvent event) -> {
```

```
            this.lastEvent = event;
        });
    }
```

Here we save away the `event` parameter into a field. This is a more subtle way of generating a side effect: assigning to variables. You may not see it directly in the output of your program, but it does change the program's state. There are limits to what Java lets you do in this regard. Take a look at the assignment to `localEvent` in this code snippet:

```
ActionEvent localEvent = null;
button.addActionListener(event -> {
    localEvent = event;
});
```

This example tries to assign the same `event` parameter into a local variable. There's no need to send me errata emails—I know this won't actually compile! That's actually a deliberate choice on behalf of the designers: an attempt to encourage people to use lambda expressions to capture values rather than capturing variables. Capturing values encourages people to write code that is free from side effects by making it harder to do so. As mentioned in Chapter 2, even though local variables don't need the `final` keyword in order to be used in lambda expressions, they still need to be *effectively* `final`.

Whenever you pass lambda expressions into the higher-order functions on the `Stream` interface, you should seek to avoid side effects. The only exception to this is the `forEach` method, which is a terminal operation.

Key Points

- Internal iteration is a way of iterating over a collection that delegates more control over the iteration to the collection.
- A `Stream` is the internal iteration analogue of an `Iterator`.
- Many common operations on collections can be performed by combining methods on `Stream` with lambda expressions.

Exercises

You can find the answers to these exercises on GitHub.

1. *Common Stream operations*. Implement the following:

a. A function that adds up numbers, i.e., `int addUp(Stream<Integer> numbers)`

b. A function that takes in artists and returns a list of strings with their names and places of origin

c. A function that takes in albums and returns a list of albums with at most three tracks

2. *Iteration*. Convert this code sample from using external iteration to internal iteration:

```
int totalMembers = 0;
for (Artist artist : artists) {
    Stream<Artist> members = artist.getMembers();
    totalMembers += members.count();
}
```

3. *Evaluation*. Take a look at the signatures of these `Stream` methods. Are they eager or lazy?

   a. `boolean anyMatch(Predicate<? super T> predicate);`

   b. `Stream<T> limit(long maxSize);`

4. *Higher-order functions*. Are these `Stream` functions higher order, and why?

   a. `boolean anyMatch(Predicate<? super T> predicate);`

   b. `Stream<T> limit(long maxSize);`

5. *Pure functions*. Are these lambda expressions side effect-free, or do they mutate state?

```
x -> x + 1
```

   Here's the example code:

```
AtomicInteger count = new AtomicInteger(0);
List<String> origins = album.musicians()
                            .forEach(musician -> count.incAndGet();)
```

   a. The lambda expression passed into `forEach` in the example.

6. Count the number of lowercase letters in a `String` (hint: look at the `chars` method on `String`).

7. Find the `String` with the largest number of lowercase letters from a `List<String>`. You can return an `Optional<String>` to account for the empty list case.

Advanced Exercises

1. Write an implementation of the `Stream` function `map` using only `reduce` and lambda expressions. You can return a `List` instead of a `Stream` if you want.
2. Write an implementation of the `Stream` function `filter` using only `reduce` and lambda expressions. Again, you can return a `List` instead of a `Stream` if you want.

CHAPTER 4
Libraries

I've talked about how to write lambda expressions but so far haven't covered the other side of the fence: how to use them. This lesson is important even if you're not writing a heavily functional library like streams. Even the simplest application is still likely to have application code that could benefit from code as data.

Another Java 8 change that has altered the way that we need to think about libraries is the introduction of `default` methods and static methods on interfaces. This change means that methods on interfaces can now have bodies and contain code.

I'll also fill in some gaps in this chapter, covering topics such as what happens when you overload methods with lambda expressions and how to use primitives. These are important things to be aware of when you're writing lambda-enabled code.

Using Lambda Expressions in Code

In Chapter 2, I described how a lambda expression is given the type of a functional interface and how this type is inferred. From the point of view of code calling the lambda expression, you can treat it identically to calling a method on an interface.

Let's look at a concrete example framed in terms of logging frameworks. Several commonly used Java logging frameworks, including `slf4j` and `log4j`, have methods that log output only when their logging level is set to a certain level or higher. So, they will have a method like `void debug(String message)` that will log `message` if the level is at `debug`.

Unfortunately, calculating the `message` to log frequently has a performance cost associated with it. Consequently, you end up with a situation in which people start explicitly calling the Boolean `isDebugEnabled` method in order to optimize this performance cost. A code sample is shown in Example 4-1. Even though a direct call to `debug` would have avoided logging the text, it still would had to call the `expensiveOperation` method

and also concatenate its output to the message `String`, so the explicit `if` check still ends up being faster.

Example 4-1. A logger using isDebugEnabled to avoid performance overhead

```
Logger logger = new Logger();
if (logger.isDebugEnabled()) {
    logger.debug("Look at this: " + expensiveOperation());
}
```

What we actually want to be able to do is pass in a lambda expression that generates a `String` to be used as the message. This expression would be called only if the `Logger` was actually at debug level or above. This approach would allow us to rewrite the previous code example to look like the code in Example 4-2.

Example 4-2. Using lambda expressions to simplify logging code

```
Logger logger = new Logger();
logger.debug(() -> "Look at this: " + expensiveOperation());
```

So how do we implement this method from within our `Logger` class? From the library point of view, we can just use the builtin `Supplier` functional interface, which has a single `get` method. We can then call `isDebugEnabled` in order to find out whether to call this method and pass the result into our `debug` method if it is enabled. The resulting code is shown in Example 4-3.

Example 4-3. The implementation of a lambda-enabled logger

```
    public void debug(Supplier<String> message) {
        if (isDebugEnabled()) {
            debug(message.get());
        }
    }
```

Calling the `get()` method in this example corresponds to calling the lambda expression that was passed into the method to be called. This approach also conveniently works with anonymous inner classes, which allows you maintain a backward-compatible API if you have consumers of your code who can't upgrade to Java 8 yet.

It's important to remember that each of the different functional interfaces can have a different name for its actual method. So, if we were using a `Predicate`, we would have to call `test`, or if we were using `Function`, we would have to call `apply`.

Primitives

You might have noticed in the previous section that we skimmed over the use of *primitive* types. In Java we have a set of parallel types—for example, `int` and `Integer`—where one is a primitive type and the other a *boxed* type. Primitive types are built into the

language and runtime environment as fundamental building blocks; boxed types are just normal Java classes that wrap up the primitives.

Because Java generics are based around *erasing* a generic parameter—in other words, pretending it's an instance of `Object`—only the boxed types can be used as generic arguments. This is why if you want a list of integer values in Java it will always be `List<Integer>` and not `List<int>`.

Unfortunately, because boxed types are objects, there is a memory overhead to them. For example, although an `int` takes 4 bytes of memory, an `Integer` takes 16 bytes. This gets even worse when you start to look at arrays of numbers, as each element of a primitive array is just the size of the primitive, while each element of a boxed array is actually an in-memory pointer to another object on the Java heap. In the worst case, this might make an `Integer[]` take up nearly six times more memory than an `int[]` of the same size.

There is also a computational overhead when converting from a primitive type to a boxed type, called *boxing*, and vice versa, called *unboxing*. For algorithms that perform lots of numerical operations, the cost of boxing and unboxing combined with the additional memory bandwidth used by allocated boxed objects can make the code significantly slower.

As a consequence of these performance overheads, the streams library differentiates between the primitive and boxed versions of some library functions. The `mapToLong` higher-order function and `ToLongFunction`, shown in Figure 4-1, are examples of this effort. Only the `int`, `long`, and `double` types have been chosen as the focus of the primitive specialization implementation in Java 8 because the impact is most noticeable in numerical algorithms.

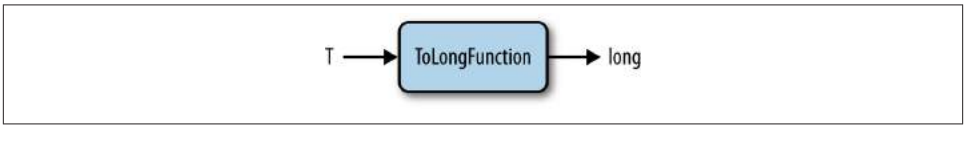

Figure 4-1. ToLongFunction

The primitive specializations have a very clear-cut naming convention. If the return type is a primitive, the interface is prefixed with `To` and the primitive type, as in `ToLongFunction` (shown in Figure 4-1). If the argument type is a primitive type, the name prefix is just the type name, as in `LongFunction` (Figure 4-2). If the higher-order function uses a primitive type, it is suffixed with `To` and the primitive type, as in `mapToLong`.

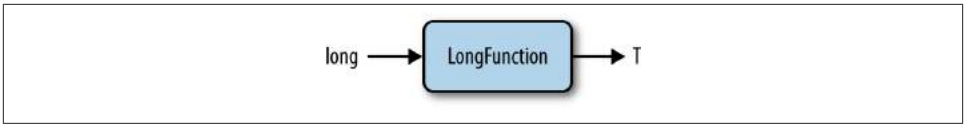

Figure 4-2. LongFunction

There are also specialized versions of `Stream` for these primitive types that prefix the type name, such as `LongStream`. In fact, methods like `mapToLong` don't return a `Stream`; they return these specialized streams. On the specialized streams, the `map` implementation is also specialized: it takes a function called `LongUnaryOperator`, visible in Figure 4-3, which maps a `long` to a `long`. It's also possible to get back from a primitive stream to a boxed stream through higher-order function variations such as `mapToObj` and the `boxed` method, which returns a stream of boxed objects such as `Stream<Long>`.

Figure 4-3. LongUnaryOperator

It's a good idea to use the primitive specialized functions wherever possible because of the performance benefits. You also get additional functionality available on the specialized streams. This allows you to avoid having to implement common functionality and to use code that better conveys the intent of numerical operations. You can see an example of how to use this functionality in Example 4-4.

Example 4-4. Using summaryStatistics to understand track length data

```
public static void printTrackLengthStatistics(Album album) {
    IntSummaryStatistics trackLengthStats
            = album.getTracks()
                   .mapToInt(track -> track.getLength())
                   .summaryStatistics();

    System.out.printf("Max: %d, Min: %d, Ave: %f, Sum: %d",
                      trackLengthStats.getMax(),
                      trackLengthStats.getMin(),
                      trackLengthStats.getAverage(),
                      trackLengthStats.getSum());
}
```

Example 4-4 prints out a summary of track length information to the console. Instead of calculating that information ourselves, we map each track to its length, using the primitive specialized `mapToInt` method. Because this method returns an `IntStream`, we

can call `summaryStatistics`, which calculates statistics such as the minimum, maximum, average, and sum values on the `IntStream`.

These values are available on all the specialized streams, such as `DoubleStream` and `LongStream`. It's also possible to calculate the individual summary statistics if you don't need all of them through the `min`, `max`, `average`, and `sum` methods, which are all also available on all three primitive specialized `Stream` variants.

Overload Resolution

It's possible in Java to *overload* methods, so you have multiple methods with the same name but different signatures. This approach poses a problem for parameter-type inference because it means that there are several types that could be inferred. In these situations `javac` will pick the *most specific* type for you. For example, the method call in Example 4-5, when choosing between the two methods in Example 4-6, prints out `String`, not `Object`.

Example 4-5. A method that could be dispatched to one of two methods

```
overloadedMethod("abc");
```

Example 4-6. Two methods that are overloaded

```
private void overloadedMethod(Object o) {
    System.out.print("Object");
}

private void overloadedMethod(String s) {
    System.out.print("String");
}
```

A `BinaryOperator` is special type of `BiFunction` for which the arguments and the return type are all the same. For example, adding two integers would be a `BinaryOperator`.

Because lambda expressions have the types of their functional interfaces, the same rules apply when passing them as arguments. We can overload a method with the `BinaryOperator` and an interface that extends it. When calling these methods, Java will infer the type of your lambda to be the most specific functional interface. For example, the code in Example 4-7 prints out `IntegerBinaryOperator` when choosing between the two methods in Example 4-8.

Example 4-7. Another overloaded method call

```
overloadedMethod((x, y) -> x + y);
```

Example 4-8. A choice between two overloaded methods

```
private interface IntegerBiFunction extends BinaryOperator<Integer> {

}

private void overloadedMethod(BinaryOperator<Integer> lambda) {
    System.out.print("BinaryOperator");
}

private void overloadedMethod(IntegerBiFunction lambda) {
    System.out.print("IntegerBinaryOperator");
}
```

Of course, when there are multiple method overloads, there isn't always a clear "most specific type." Take a look at Example 4-9.

Example 4-9. A compile failure due to overloaded methods

```
overloadedMethod((x) -> true);

private interface IntPredicate  {
    public boolean test(int value);
}

private void overloadedMethod(Predicate<Integer> predicate) {
    System.out.print("Predicate");
}

private void overloadedMethod(IntPredicate predicate) {
    System.out.print("IntPredicate");
}
```

The lambda expression passed into `overloadedMethod` is compatible with both a normal `Predicate` and the `IntPredicate`. There are method overloads for each of these options defined within this code block. In this case, `javac` will fail to compile the example, complaining that the lambda expression is an ambiguous method call: `IntPredicate` doesn't extend any `Predicate`, so the compiler isn't able to infer that it's more specific.

The way to fix these situations is to cast the lambda expression to either `IntPredicate` or `Predicate<Integer>`, depending upon which behavior you want to call. Of course, if you've designed the library yourself, you might conclude that this is a code smell and you should start renaming your overloaded methods.

In summary, the parameter types of a lambda are inferred from the *target type*, and the inference follows these rules:

- If there is a single possible target type, the lambda expression infers the type from the corresponding argument on the functional interface.
- If there are several possible target types, the most specific type is inferred.

- If there are several possible target types and there is no most specific type, you must manually provide a type.

@FunctionalInterface

Although I talked about the criteria for what a functional interface actually is back in Chapter 2, I haven't yet mentioned the `@FunctionalInterface` annotation. This is an annotation that should be applied to any interface that is intended to be used as a functional interface.

What does that really mean? Well, there are some interfaces in Java that have only a single method but aren't normally meant to be implemented by lambda expressions. For example, they might assume that the object has internal state and be interfaces with a single method only coincidentally. A couple of good examples are `java.lang.Comparable` and `java.io.Closeable`.

If a class is `Comparable`, it means there is a defined order between instances, such as alphabetical order for strings. You don't normally think about functions themselves as being comparable objects because they lack fields and state, and if there are no fields and no state, what is there to sensibly compare?

For an object to be `Closeable` it must hold an open resource, such as a file handle that needs to be closed at some point in time. Again, the interface being called cannot be a pure function because closing a resource is really another example of mutating state.

In contrast to `Closeable` and `Comparable`, all the new interfaces introduced in order to provide `Stream` interoperability are expected to be implemented by lambda expressions. They are really there to bundle up blocks of code as data. Consequently, they have the `@FunctionalInterface` annotation applied.

Using the annotation compels `javac` to actually check whether the interface meets the criteria for being a functional interface. If the annotation is applied to an `enum`, `class`, or `annotation`, or if the type is an interface with more than one single abstract method, then `javac` will generate an error message. This is quite helpful for being able to catch errors easily when refactoring your code.

Binary Interface Compatibility

As you saw in Chapter 3, one of the biggest API changes in Java 8 is to the collections library. As Java has evolved, it has maintained backward binary compatibility. In practical terms, this means that if you compiled a library or application with Java 1 through 7, it'll run out of the box in Java 8.

Of course, there are still bugs from time to time, but compared to many other programming platforms, binary compatibility has been viewed as a key Java strength. Barring the introduction of a new keyword, such as `enum`, there has also been an effort to maintain backward source compatibility. Here the guarantee is that if you've got source code in Java 1-7, it'll compile in Java 8.

These guarantees are really hard to maintain when you're changing such a core library component as the collections library. As a thought exercise, consider a concrete example. The `stream` method was added to the `Collection` interface in Java 8, which means that any class that implements `Collection` must also have this method on it. For core library classes, this problem can easily be solved by implementing that method (e.g., adding a `stream` method to `ArrayList`).

Unfortunately, this change still breaks binary compatibility because it means that any class outside of the JDK that implements `Collection`—say, `MyCustomList`—must also have implemented the `stream` method. In Java 8 `MyCustomList` would no longer compile, and even if you had a compiled version when you tried to load `MyCustomList` into a JVM, it would result in an exception being thrown by your `ClassLoader`.

This nightmare scenario of all third-party collections libraries being broken has been averted, but it did require the introduction of a new language concept: *default methods*.

Default Methods

So you've got your new `stream` method on `Collection`; how do you allow `MyCustomList` to compile without ever having to know about its existence? The Java 8 approach to solving the problem is to allow `Collection` to say, "If any of my children don't have a `stream` method, they can use this one." These methods on an interface are called *default* methods. They can be used on any interface, functional or not.

Another default method that has been added is the `forEach` method on `Iterable`, which provides similar functionality to the `for` loop but lets you use a lambda expression as the body of the loop. Example 4-10 shows how this could be implemented in the JDK.

Example 4-10. An example default method, showing how forEach might be implemented

```
default void forEach(Consumer<? super T> action) {
    for (T t : this) {
        action.accept(t);
    }
}
```

Now that you're familiar with the idea that you can use lambda expressions by just calling methods on interfaces, this example should look pretty simple. It uses a regular `for` loop to iterate over the underlying `Iterable`, calling the `accept` method with each value.

If it's so simple, why mention it? The important thing is that new `default` keyword right at the beginning of the code snippet. That tells `javac` that you really want to add a method to an interface. Other than the addition of a new keyword, `default` methods also have slightly different inheritance rules to regular methods.

The other big difference is that, unlike classes, interfaces don't have instance fields, so `default` methods can modify their child classes only by calling methods on them. This helps you avoid making assumptions about the implementation of their children.

Default Methods and Subclassing

There are some subtleties about the way that `default` methods override and can be overridden by other methods. Let's look the simplest case to begin with: no overriding. In Example 4-11, our `Parent` interface defines a `welcome` method that sends a message when called. The `ParentImpl` class doesn't provide an implementation of `welcome`, so it inherits the `default` method.

Example 4-11. The Parent interface; the welcome method is a default

```
public interface Parent {

    public void message(String body);

    public default void welcome() {
        message("Parent: Hi!");
    }

    public String getLastMessage();

}
```

When we come to call this code, in Example 4-12, the `default` method is called and our assertion passes.

Example 4-12. Using the default method from client code

```
@Test
public void parentDefaultUsed() {
    Parent parent = new ParentImpl();
    parent.welcome();
    assertEquals("Parent: Hi!", parent.getLastMessage());
}
```

Now we can extend `Parent` with a `Child` interface, whose code is listed in Example 4-13. `Child` implements its own `default welcome` method. As you would intuitively expect, the `default` method on `Child` overrides the `default` method on `Parent`. In this example, again, the `ChildImpl` class doesn't provide an implementation of `welcome`, so it inherits the `default` method.

Example 4-13. Child interface that extends Parent

```
public interface Child extends Parent {

    @Override
    public default void welcome() {
        message("Child: Hi!");
    }

}
```

You can see the class hierarchy at this point in Figure 4-4.

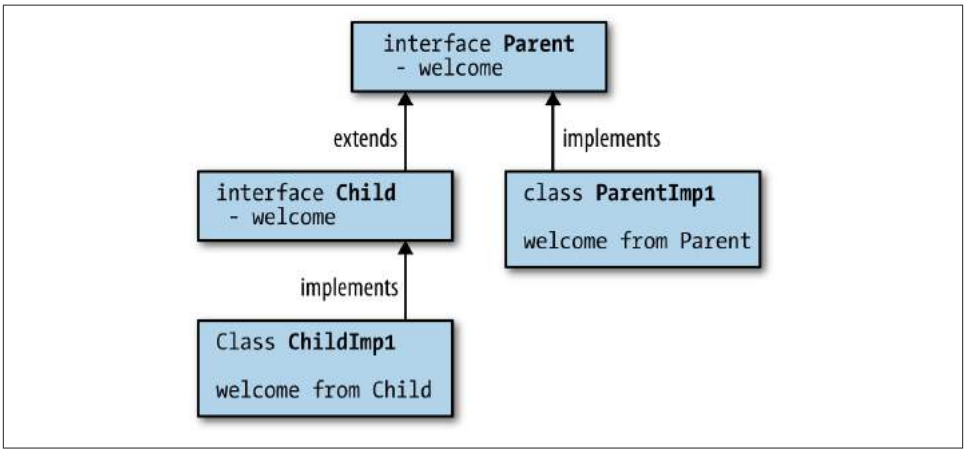

Figure 4-4. A diagram showing the inheritance hierarchy at this point

Example 4-14 calls this interface and consequently ends up sending the string `"Child: Hi!"`.

Example 4-14. Client code that calls our Child interface

```
@Test
public void childOverrideDefault() {
    Child child = new ChildImpl();
    child.welcome();
    assertEquals("Child: Hi!", child.getLastMessage());
}
```

Now the `default` method is a virtual method—that is, the opposite of a static method. What this means is that whenever it comes up against competition from a class method, the logic for determining which override to pick always chooses the class. A simple example of this is shown in Examples 4-15 and 4-16, where the `welcome` method of `OverridingParent` is chosen over that of `Parent`.

Example 4-15. A parent class that overrides the default implementation of welcome

```
public class OverridingParent extends ParentImpl {

    @Override
    public void welcome() {
        message("Class Parent: Hi!");
    }

}
```

Example 4-16. An example of a concrete method beating a default method

```
@Test
public void concreteBeatsDefault() {
    Parent parent = new OverridingParent();
    parent.welcome();
    assertEquals("Class Parent: Hi!", parent.getLastMessage());
}
```

Here's a situation, presented in Example 4-18, in which you might not expect the concrete class to override the `default` method. `OverridingChild` inherits both the `welcome` method from `Child` and the `welcome` method from `OverridingParent` and doesn't do anything itself. `OverridingParent` is chosen despite `OverridingChild` (the code in Example 4-17), being a more specific type because it's a concrete method from a class rather than a `default` method (see Figure 4-5).

Example 4-17. Again, our child interface overrides the default welcome method

```
public class OverridingChild extends OverridingParent implements Child {

}
```

Example 4-18. An example of a concrete method beating a default method that is more specific

```
@Test
public void concreteBeatsCloserDefault() {
    Child child = new OverridingChild();
    child.welcome();
    assertEquals("Class Parent: Hi!", child.getLastMessage());
}
```

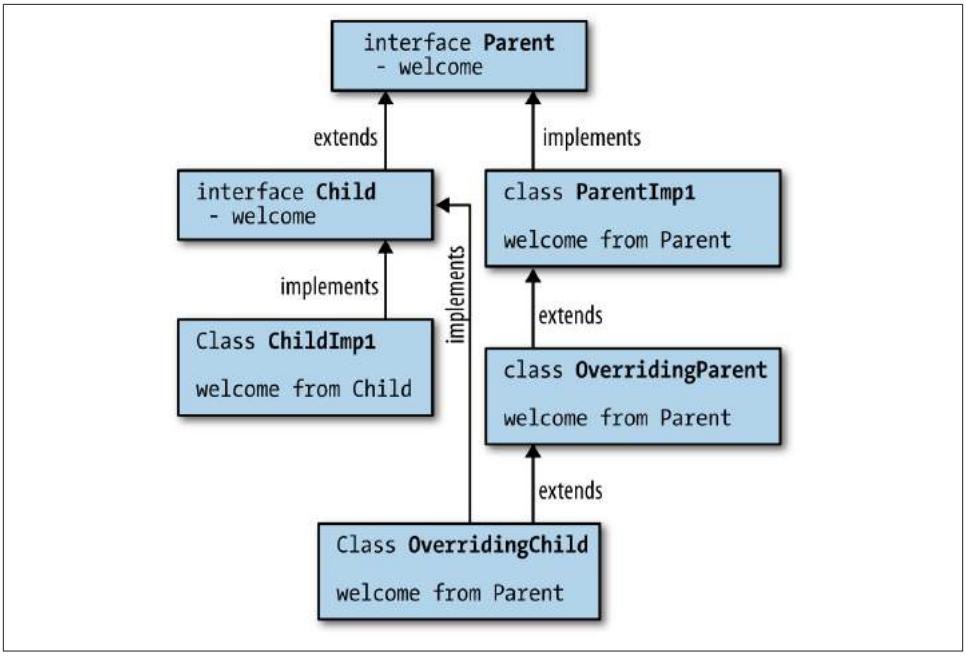

Figure 4-5. A diagram showing the complete inheritance hierarchy

Put simply: *class wins*. The motivation for this decision is that `default` methods are designed primarily to allow binary compatible API evolution. Allowing classes to win over any `default` methods simplifies a lot of inheritance scenarios.

Suppose we had a custom list implementation called `MyCustomList` and had implemented a custom `addAll` method, and the new `List` interface provided a `default addAll` that delegated to the `add` method. If the `default` method wasn't guaranteed to be overridden by this `addAll` method, we could break the existing implementation.

Multiple Inheritance

Because interfaces are subject to multiple inheritance, it's possible to get into situations where two interfaces both provide `default` methods with the same signature. Here's an example in which both a `Carriage` and a `Jukebox` provide a method to `rock`—in each case, for different purposes. We also have a `MusicalCarriage`, which is both a `Jukebox` (Example 4-19) and a `Carriage` (Example 4-20) and tries to inherit the `rock` method.

Example 4-19. Jukebox

```
public interface Jukebox {

    public default String rock() {
```

```
        return "... all over the world!";
    }

}
```

Example 4-20. Carriage

```
public interface Carriage {

    public default String rock() {
        return "... from side to side";
    }

}

    public class MusicalCarriage implements Carriage, Jukebox {
    }
```

Because it's not clear to `javac` which method it should inherit, this will just result in the compile error `class MusicalCarriage inherits unrelated defaults for rock() from types Carriage and Jukebox`. Of course, it's possible to resolve this by implementing the `rock` method, shown in Example 4-21.

Example 4-21. Implementing the rock method

```
public class MusicalCarriage
        implements Carriage, Jukebox {

    @Override
    public String rock() {
        return Carriage.super.rock();
    }

}
```

This example uses the enhanced `super` syntax in order to pick `Carriage` as its preferred `rock` implementation. Previously, `super` acted as a reference to the parent class, but by using the `InterfaceName.super` variant it's possible to specify a method from an inherited interface.

The Three Rules

If you're ever unsure of what will happen with `default` methods or with multiple inheritance of behavior, there are three simple rules for handling conflicts:

1. Any class wins over any interface. So if there's a method with a body, or an abstract declaration, in the superclass chain, we can ignore the interfaces completely.

2. Subtype wins over supertype. If we have a situation in which two interfaces are competing to provide a `default` method and one interface extends the other, the subclass wins.
3. *No rule 3*. If the previous two rules don't give us the answer, the subclass must either implement the method or declare it `abstract`.

Rule 1 is what brings us compatibility with old code.

Tradeoffs

These changes raise a bunch of issues regarding what an interface really is in Java 8, as you can define methods with code bodies on them. This means that interfaces now provide a form of multiple inheritance that has previously been frowned upon and whose removal has been considered a usability advantage of Java over C++.

No language feature is always good or always bad. Many would argue that the real issue is multiple inheritance of state rather than just blocks of code, and as `default` methods avoid multiple inheritance of state, they avoid the worst pitfalls of multiple inheritance in C++.

It can also be very tempting to try and work around these limitations. Blog posts have already cropped up trying to implement full-on traits with multiple inheritance of state as well as `default` methods. Trying to hack around the deliberate restrictions of Java 8 puts us back into the old pitfalls of C++.

It's also pretty clear that there's still a distinction between interfaces and abstract classes. Interfaces give you multiple inheritance but no fields, while abstract classes let you inherit fields but you don't get multiple inheritance. When modeling your problem domain, you need to think about this tradeoff, which wasn't necessary in previous versions of Java.

Static Methods on Interfaces

We've seen a lot of calling of `Stream.of` but haven't gotten into its details yet. You may recall that `Stream` is an interface, but this is a static method on an interface. This is another new language change that has made its way into Java 8, primarily in order to help library developers, but with benefits for day-to-day application developers as well.

An idiom that has accidentally developed over time is ending up with classes full of static methods. Sometimes a class can be an appropriate location for utility code, such as the `Objects` class introduced in Java 7 that contained functionality that wasn't specific to any particular class.

Of course, when there's a good semantic reason for a method to relate to a concept, it should always be put in the same class or interface rather than hidden in a utility class to the side. This helps structure your code in a way that's easier for someone reading it to find the relevant method.

For example, if you want to create a simple `Stream` of values, you would expect the method to be located on `Stream`. Previously, this was impossible, and the addition of a very interface-heavy API in terms of `Stream` finally motivated the addition of static methods on interfaces.

There are other methods on `Stream` and its primitive specialized variants. Specifically, `range` and `iterate` give us other ways of generating our own streams.

Optional

Something I've glossed over so far is that `reduce` can come in a couple of forms: the one we've seen, which takes an initial value, and another variant, which doesn't. When the initial value is left out, the first call to the reducer uses the first two elements of the `Stream`. This is useful if there's no sensible initial value for a `reduce` operation and will return an instance of `Optional`.

`Optional` is a new core library data type that is designed to provide a better alternative to `null`. There's quite a lot of hatred for the old `null` value. Even the man who invented the concept, Tony Hoare, described it as "my billion-dollar mistake." That's the trouble with being an influential computer scientist—you can make a billion-dollar mistake without even seeing the billion dollars yourself!

`null` is often used to represent the absence of a value, and this is the use case that `Optional` is replacing. The problem with using `null` in order to represent absence is the dreaded `NullPointerException`. If you refer to a variable that is `null`, your code blows up. The goal of `Optional` is twofold. First, it encourages the coder to make appropriate checks as to whether a variable is `null` in order to avoid bugs. Second, it documents values that are expected to be absent in a class's API. This makes it easier to see where the bodies are buried.

Let's take a look at the API for `Optional` in order to get a feel for how to use it. If you want to create an `Optional` instance from a value, there is a factory method called `of`. The `Optional` is now a container for this value, which can be pulled out with `get`, as shown in Example 4-22.

Example 4-22. Creating an Optional from a value

```
Optional<String> a = Optional.of("a");
assertEquals("a", a.get());
```

Because an `Optional` may also represent an absent value, there's also a factory method called `empty`, and you can convert a nullable value into an `Optional` using the `ofNullable` method. Both of these are shown in Example 4-23, along with the use of the `isPresent` method (which indicates whether the `Optional` is holding a value).

Example 4-23. Creating an empty Optional and checking whether it contains a value

```
Optional emptyOptional = Optional.empty();
Optional alsoEmpty = Optional.ofNullable(null);

assertFalse(emptyOptional.isPresent());

// a is defined above
assertTrue(a.isPresent());
```

One approach to using `Optional` is to guard any call to `get()` by checking `isPresent()`. A neater approach is to call the `orElse` method, which provides an alternative value in case the `Optional` is empty. If creating an alternative value is computationally expensive, the `orElseGet` method should be used. This allows you to pass in a `Supplier` that is called only if the `Optional` is genuinely empty. Both of these methods are demonstrated in Example 4-24.

Example 4-24. Using orElse and orElseGet

```
assertEquals("b", emptyOptional.orElse("b"));
assertEquals("c", emptyOptional.orElseGet(() -> "c"));
```

Not only is `Optional` used in new Java 8 APIs, but it's also just a regular class that you can use yourself when writing domain classes. This is definitely something to think about when trying to avoid nullness-related bugs such as uncaught exceptions.

Key Points

- A significant performance advantage can be had by using primitive specialized lambda expressions and streams such as `IntStream`.
- Default methods are methods with bodies on interfaces prefixed with the keyword `default`.
- The `Optional` class lets you avoid using `null` by modeling situations where a value may not be present.

Exercises

1. Given the `Performance` interface in Example 4-25, add a method called `getAllMusicians` that returns a `Stream` of the artists performing and, in the case of groups, any musicians who are members of those groups. For example, if `getMusicians` returns *The Beatles*, then you should return *The Beatles* along with Lennon, McCartney, and so on.

   Example 4-25. An interface denoting the concept of a musical performance

   ```
   /** A Performance by some musicians - e.g., an Album or Gig. */
   public interface Performance {

       public String getName();

       public Stream<Artist> getMusicians();

   }
   ```

2. Based on the resolution rules described earlier, can you ever override `equals` or `hashCode` in a `default` method?
3. Take a look at the `Artists` domain class in Example 4-26, which represents a group of artists. Your assignment is to refactor the `getArtist` method in order to return an `Optional<Artist>`. It contains an element if the index is within range and is an empty `Optional` otherwise. Remember that you also need to refactor the `getArtistName` method, and it should retain the same behavior.

Example 4-26. The Artists domain class, which represents more than one Artist

```
public class Artists {

    private List<Artist> artists;

    public Artists(List<Artist> artists) {
        this.artists = artists;
    }

    public Artist getArtist(int index) {
        if (index < 0 || index >= artists.size()) {
            indexException(index);
        }
        return artists.get(index);
    }

    private void indexException(int index) {
        throw new IllegalArgumentException(index +
                                           " doesn't correspond to an Artist");
    }
```

```
    public String getArtistName(int index) {
        try {
            Artist artist = getArtist(index);
            return artist.getName();
        } catch (IllegalArgumentException e) {
            return "unknown";
        }
    }

}
```

Open Exercises

1. Look through your work code base or an open source project you're familiar with and try to identify classes that have just static methods that could be moved to static methods on interfaces. It might be worth discussing with your colleagues whether they agree or disagree with you.

CHAPTER 5

Advanced Collections and Collectors

There's a lot more to the collections library changes than I covered in Chapter 3. It's time to cover some of the more advanced collections changes, including the new `Collector` abstraction. I'll also introduce method references, which are a way of using existing code in lambda expressions with little to no ceremony. They pay huge dividends when it comes to writing `Collection`-heavy code. More advanced topics within the collections library will also be covered, such as element ordering within streams and other useful API changes.

Method References

A common idiom you may have noticed is the creation of a lambda expression that calls a method on its parameter. If we want a lambda expression that gets the name of an artist, we would write the following:

```
artist -> artist.getName()
```

This is such a common idiom that there's actually an abbreviated syntax for this that lets you reuse an existing method, called a *method reference*. If we were to write the previous lambda expression using a method reference, it would look like this:

```
Artist::getName
```

The standard form is `Classname::methodName`. Remember that even though it's a method, you don't need to use brackets because you're not actually calling the method. You're providing the equivalent of a lambda expression that can be called in order to call the method. You can use method references in the same places as lambda expressions.

You can also call constructors using the same abbreviated syntax. If you were to use a lambda expression to create an `Artist`, you might write:

```
(name, nationality) -> new Artist(name, nationality)
```

We can also write this using method references:

```
Artist::new
```

This code is not only shorter but also a lot easier to read. `Artist::new` immediately tells you that you're creating a new `Artist` without your having to scan the whole line of code. Another thing to notice here is that method references automatically support multiple parameters, as long as you have the right functional interface.

It's also possible to create arrays using this method. Here is how you would create a `String` array:

```
String[]::new
```

We'll be using method references from this point onward where appropriate, so you'll be seeing a lot more examples very soon. When we were first exploring the Java 8 changes, a friend of mine said that method references "feel like cheating." What he meant was that, having looked at how we can use lambda expressions to pass code around as if it were data, it felt like cheating to be able to reference a method directly.

It's OK—it's not cheating! You have to bear in mind that every time you write a lambda expression that looks like `x -> foo(x)`, it's really doing the same thing as just the method `foo` on its own. All method references do is provide a simpler syntax that takes advantage of this fact.

Element Ordering

One topic I haven't discussed so far that pertains to collections is how elements are ordered in streams. You might be familiar with the concept that some types of `Collection`, such as `List`, have a defined order, and collections like `HashSet` don't. The situation with ordering becomes a little more complex with `Stream` operations.

A `Stream` intuitively presents an order because each element is operated upon, or encountered, in turn. We call this the *encounter order*. How the encounter order is defined depends on both the source of the data and the operations performed on the `Stream`.

When you create a `Stream` from a collection with a defined order, the `Stream` has a defined encounter order. As a consequence, Example 5-1 will always pass.

Example 5-1. The ordering assumption in this test will always work

```
List<Integer> numbers = asList(1, 2, 3, 4);

List<Integer> sameOrder = numbers.stream()
                                 .collect(toList());
assertEquals(numbers, sameOrder);
```

If there's no defined order to begin, the `Stream` produced by that source doesn't have a defined order. A `HashSet` is an example of a collection without a defined ordering, and because of that Example 5-2 isn't guaranteed to pass.

Example 5-2. The ordering assumption here isn't guaranteed

```
Set<Integer> numbers = new HashSet<>(asList(4, 3, 2, 1));

List<Integer> sameOrder = numbers.stream()
                                 .collect(toList());

// This may not pass
assertEquals(asList(4, 3, 2, 1), sameOrder);
```

The purpose of streams isn't just to convert from one collection to another; it's to be able to provide a common set of operations over data. These operations may create an encounter order where there wasn't one to begin with. Consider the code presented in Example 5-3.

Example 5-3. Creating an encounter order

```
Set<Integer> numbers = new HashSet<>(asList(4, 3, 2, 1));

List<Integer> sameOrder = numbers.stream()
                                 .sorted()
                                 .collect(toList());

assertEquals(asList(1, 2, 3, 4), sameOrder);
```

The encounter order is propagated across intermediate operations if it exists; for example, if we try to `map` values and there's a defined encounter order, then that encounter order will be preserved. If there's no encounter order on the input `Stream`, there's no encounter order on the output `Stream`. Consider the two snippets of code in Example 5-4. We can only make the weaker `hasItem` assertions on the `HashSet` example because the lack of a defined encounter order from `HashSet` continues through the `map`.

Example 5-4. The ordering assumption in this test will always work

```
List<Integer> numbers = asList(1, 2, 3, 4);

List<Integer> stillOrdered = numbers.stream()
                                    .map(x -> x + 1)
                                    .collect(toList());

// Reliable encounter ordering
assertEquals(asList(2, 3, 4, 5), stillOrdered);

Set<Integer> unordered = new HashSet<>(numbers);

List<Integer> stillUnordered = unordered.stream()
                                        .map(x -> x + 1)
```

```
                                        .collect(toList());

    // Can't assume encounter ordering
    assertThat(stillUnordered, hasItem(2));
    assertThat(stillUnordered, hasItem(3));
    assertThat(stillUnordered, hasItem(4));
    assertThat(stillUnordered, hasItem(5));
```

Some operations are more expensive on ordered streams. This problem can be solved by eliminating ordering. To do so, call the stream's `unordered` method. Most operations, however, such as `filter`, `map`, and `reduce`, can operate very efficiently on ordered streams.

This can cause unexpected behavior, for example, `forEach` provides no guarantees as to encounter order if you're using parallel streams. (This will be discussed in more detail in Chapter 6.) If you require an ordering guarantee in these situations, then `forEachOrdered` is your friend!

Enter the Collector

Earlier, we used the `collect(toList())` idiom in order to produce lists out of streams. Obviously, a `List` is a very natural value to want to produce from a `Stream`, but it's not the only value that you might want to compute. Perhaps you want to generate a `Map` or a `Set`. Maybe you think it's worth having a domain class that abstracts the concept you want?

You've already learned that you can tell just from the signature of a `Stream` method whether it's an eagerly evaluated terminal operation that can be used to produce a value. A `reduce` operation can be very suitable for this purpose. Sometimes you want to go further than `reduce` allows, though.

Enter the *collector*, a general-purpose construct for producing complex values from streams. These can be used with any `Stream` by passing them into the `collect` method.

The standard library provides a bunch of useful collectors out of the box, so let's look at those first. In the code examples throughout this chapter the collectors are statically imported from the `java.util.stream.Collectors` class.

Into Other Collections

Some collectors just build up other collections. You've already seen the `toList` collector, which produces `java.util.List` instances. There's also a `toSet` collector and a `toCollection` collector, which produce instances of `Set` and `Collection`. I've talked a lot so far about chaining `Stream` operations, but there are still times when you'll want to produce a `Collection` as a final value—for example:

- When passing your collection to existing code that is written to use collections
- When creating a final value at the end of a chain of collections
- When writing test case asserts that operate on a concrete collection

Normally when we create a collection, we specify the concrete type of the collection by calling the appropriate constructor:

```
List<Artist> artists = new ArrayList<>();
```

But when you're calling `toList` or `toSet`, you don't get to specify the concrete implementation of the `List` or `Set`. Under the hood, the streams library is picking an appropriate implementation for you. Later in this book I'll talk about how you can use the streams library to perform data parallel operations; collecting the results of parallel operations can require a different type of `Set` to be produced than if there were no requirement for thread safety.

It might be the case that you wish to `collect` your values into a `Collection` of a specific type if you require that type later. For example, perhaps you want to use a `TreeSet` instead of allowing the framework to determine what type of `Set` implementation you get. You can do that using the `toCollection` collector, which takes a function to build the collection as its argument (see Example 5-5).

Example 5-5. Collecting into a custom collection using toCollection

```
stream.collect(toCollection(TreeSet::new));
```

To Values

It's also possible to collect into a single value using a collector. There are `maxBy` and `minBy` collectors that let you obtain a single value according to some ordering. Example 5-6 shows how to find the band with the most members. It defines a lambda expression that can map an artist to the number of members. This is then used to define a comparator that is passed into the `maxBy` collector.

Example 5-6. Finding the band with the most members

```
public Optional<Artist> biggestGroup(Stream<Artist> artists) {
    Function<Artist,Long> getCount = artist -> artist.getMembers().count();
    return artists.collect(maxBy(comparing(getCount)));
}
```

There's also a `minBy`, which does what it says on the tin.

There are also collectors that implement common numerical operations. Let's take a look at these by writing a collector to find the average number of tracks on an album, as in Example 5-7.

Example 5-7. Finding the average number of tracks for a list of albums

```
public double averageNumberOfTracks(List<Album> albums) {
    return albums.stream()
                 .collect(averagingInt(album -> album.getTrackList().size()));
}
```

As usual, we kick off our pipeline with the `stream` method and `collect` the results. We then call the `averagingInt` method, which takes a lambda expression in order to convert each element in the `Stream` into an `int` before averaging the values. There are also overloaded operations for the `double` and `long` types, which let you convert your element into these type of values.

Back in "Primitives" on page 42, we talked about how the primitive specialized variants of streams, such as `IntStream`, had additional functionality for numerical operations. In fact, there are also a group of collectors that offer similar functionality, in the vein of `averagingInt`. You can add up the values using `summingInt` and friends. `SummaryStatistics` is collectible using `summarizingInt` and its combinations.

Partitioning the Data

Another common operation that you might want to do with a `Stream` is partition it into two collections of values. For example, if you've got a `Stream` of artists, then you might wish to get all the artists who are solo artists—that is, who have no fellow band members—and all the artists who are bands. One approach to doing this is to perform two different filters, one looking for solo artists and the other for bands.

This approach has a couple of downsides, though. First, you'll need two streams in order to perform these two stream operations. Second, if you've got a long sequence of operations leading up to your filters, these will need to be performed twice over each stream. This also doesn't result in clean code.

Consequently, there is a collector, `partitioningBy`, that takes a stream and partitions its contents into two groups (see Figure 5-1). It uses a `Predicate` to determine whether an element should be part of the true group or the false group and returns a `Map` from `Boolean` to a `List` of values. So, the `Predicate` returns `true` for all the values in the `true List` and `false` for the other `List`.

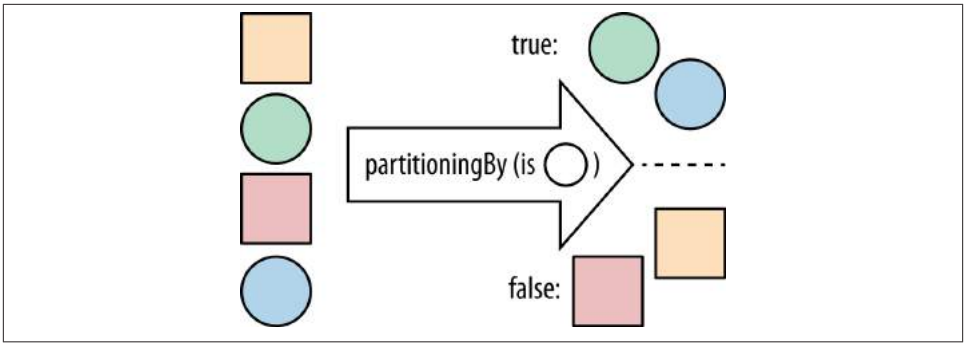

Figure 5-1. The partitioningBy collector

We can use these features to split out bands (artists with more than one member) from solo artists. In this case, our partitioning function tells us whether the artist is a solo act. Example 5-8 provides an implementation.

Example 5-8. Partitioning a stream of artists into bands and solo artists

```
public Map<Boolean, List<Artist>> bandsAndSolo(Stream<Artist> artists) {
    return artists.collect(partitioningBy(artist -> artist.isSolo()));
}
```

We can also write this using method references, as demonstrated in Example 5-9.

Example 5-9. Partitioning up a stream of artists into bands and solo artists using a method reference

```
public Map<Boolean, List<Artist>> bandsAndSoloRef(Stream<Artist> artists) {
    return artists.collect(partitioningBy(Artist::isSolo));
}
```

Grouping the Data

There's a natural way to generalize partitioning through altering the grouping operation. It's more general in the sense that instead of splitting up your data into `true` and `false` groups, you can use whatever values you want. Perhaps some code has given you a `Stream` of albums and you want to group them by the name of their main musician. You might write some code like Example 5-10.

Example 5-10. Grouping albums by their main artist

```
public Map<Artist, List<Album>> albumsByArtist(Stream<Album> albums) {
    return albums.collect(groupingBy(album -> album.getMainMusician()));
}
```

As with the other examples, we're calling `collect` on the `Stream` and passing in a `Collector`. Our `groupingBy` collector (Figure 5-2) takes a `classifier` function in order

to partition the data, just like the `partitioningBy` collector took a `Predicate` to split it up into `true` and `false` values. Our classifier is a `Function`—the same type that we use for the common `map` operation.

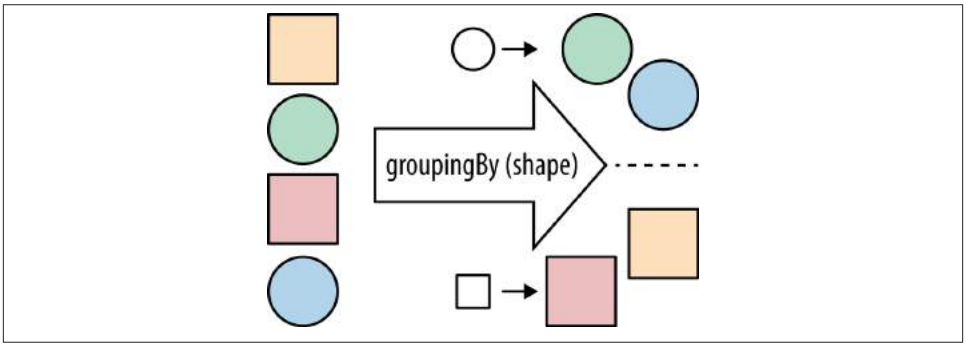

Figure 5-2. The groupingBy collector

You might be familiar with `group by` from using SQL; here we have a method with a similar concept, but implemented in the idioms of the streams library.

Strings

A very common reason for collecting streams of data is to generate strings at the end. Let's suppose that we want to put together a formatted list of names of the artists involved in an album. So, for example, if our input album is *Let It Be*, then we're expecting our output to look like `"[George Harrison, John Lennon, Paul McCartney, Ringo Starr, The Beatles]"`.

If we were to implement this before Java 8, we might have come up with something like Example 5-11. Here, we use a `StringBuilder` to accumulate the values, iterating over the list. At each step, we pull out the names of the artists and add them to the `StringBuilder`.

Example 5-11. Formatting artist names using a for loop

```
StringBuilder builder = new StringBuilder("[");
for (Artist  artist : artists) {
    if (builder.length() > 1)
        builder.append(", ");

    String name = artist.getName();
    builder.append(name);
}
```

```
builder.append("]");
String result = builder.toString();
```

Of course, this isn't particularly great code. It's pretty hard to see what it's doing without walking through it step by step. With Java 8 we can write Example 5-12, which makes our intent much clearer using streams and collectors.

Example 5-12. Formatting artist names using streams and collectors

```
String result =
    artists.stream()
              .map(Artist::getName)
              .collect(Collectors.joining(", ", "[", "]"));
```

Here, we use a `map` to extract the artists' names and then `collect` the `Stream` using `Collectors.joining`. This method is a convenience for building up strings from streams. It lets us provide a delimiter (which goes between elements), a prefix for our result, and a suffix for the result.

Composing Collectors

Although the collectors we've seen so far are quite powerful, they become significantly more so when composed with other collectors.

Previously we grouped albums by their main artist; now let's consider the problem of counting the number of albums for each artist. A simple approach would be to apply the previous grouping and then count the values. You can see how that works out in Example 5-13.

Example 5-13. A naive approach to counting the number of albums for each artist

```
Map<Artist, List<Album>> albumsByArtist
    = albums.collect(groupingBy(album -> album.getMainMusician()));

Map<Artist, Integer>  numberOfAlbums = new HashMap<>();
for(Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
    numberOfAlbums.put(entry.getKey(), entry.getValue().size());
}
```

Hmm, it might have sounded like a simple approach, but it got a bit messy. This code is also imperative and doesn't automatically parallelize.

What we want here is actually another collector that tells `groupingBy` that instead of building up a `List` of albums for each artist, it should just count them. Conveniently, this is already in the core library and is called `counting`. So, we can rewrite the example into Example 5-14.

Example 5-14. Using collectors to count the number of albums for each artist

```
public Map<Artist, Long> numberOfAlbums(Stream<Album> albums) {
    return albums.collect(groupingBy(album -> album.getMainMusician(),
                                     counting()));
}
```

This form of `groupingBy` divides elements into *buckets*. Each bucket gets associated with the key provided by the classifier function: `getMainMusician`. The `groupingBy` operation then uses the downstream collector to collect each bucket and makes a map of the results.

Let's consider another example, in which instead of building up a grouping of albums, we just want their names. Again, one approach is to take our original collector and then fix up the resulting values in the `Map`. Example 5-15 shows how we might do that.

Example 5-15. A naive approach to finding the names of every album that an artist has produced

```
public Map<Artist, List<String>> nameOfAlbumsDumb(Stream<Album> albums) {
    Map<Artist, List<Album>> albumsByArtist =
            albums.collect(groupingBy(album ->album.getMainMusician()));

    Map<Artist, List<String>>  nameOfAlbums = new HashMap<>();
    for(Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
        nameOfAlbums.put(entry.getKey(), entry.getValue()
                                            .stream()
                                            .map(Album::getName)
                                            .collect(toList()));
    }
    return nameOfAlbums;
}
```

Again, we can produce nicer, faster, and easier-to-parallelize code using another collector. We already know that we can group our albums by the main artist using the `groupingBy` collector, but that would output a `Map<Artist, List<Album>>`. Instead of associating a list of albums with each `Artist`, we want to associate a list of strings, each of which is the name of an album.

In this case, what we're really trying to do is perform a `map` operation on the list from the `Artist` to the album name. We can't just use the `map` method on streams because this list is created by the `groupingBy` collector. We need a way of telling the `groupingBy` collector to `map` its list values as it's building up the result.

Each collector is a recipe for building a final value. What we really want is a recipe to give to our recipe—*another* collector. Thankfully, the boffins at Oracle have thought of this use case and provided a collector called `mapping`.

The `mapping` collector allows you to perform a `map`-like operation over your collector's container. You also need to tell your `mapping` collector what collection it needs to store the results in, which you can do with the `toList` collector. It's turtles, I mean collectors, all the way down!

Just like `map`, this takes an implementation of `Function`. If we refactor our code to use a second collector, we end up with Example 5-16.

Example 5-16. Using collectors to find the names of every album that an artist has produced

```
public Map<Artist, List<String>> nameOfAlbums(Stream<Album> albums) {
    return albums.collect(groupingBy(Album::getMainMusician,
                                     mapping(Album::getName, toList())));
}
```

In both of these cases, we've used a second collector in order to collect a subpart of the final result. These collectors are called *downstream* collectors. In the same way that a collector is a recipe for building a final value, a downstream collector is a recipe for building a part of that value, which is then used by the main collector. The way you can compose collectors like this makes them an even more powerful component in the streams library.

The primitive specialized functions, such as `averagingInt` or `summarizingLong`, are actually duplicate functionality over calling the method on the specialized stream themselves. The real motivation for them to exist is to be used as downstream collectors.

Refactoring and Custom Collectors

Although the built-in Java collectors are good building blocks for common operations around streams, the collector framework is very generic. There is nothing special or magic about the ones that ship with the JDK, and you can build your own collectors very simply. That's what we'll look at now.

You may recall when we looked at strings that we could write our example in Java 7, albeit inelegantly. Let's take this example and slowly refactor it into a proper `String`-joining collector. There's no need for you to use this code—the JDK provides a perfectly good `joining` collector—but it is quite an instructive example both of how custom collectors work and of how to refactor legacy code into Java 8.

Example 5-17 is a reminder of our Java 7 `String`-joining example.

Example 5-17. Using a for loop and a StringBuilder to pretty-print the names of artists

```
StringBuilder builder = new StringBuilder("[");
for (Artist  artist : artists) {
    if (builder.length() > 1)
        builder.append(", ");
```

```
    String name = artist.getName();
    builder.append(name);
}
builder.append("]");
String result = builder.toString();
```

It's pretty obvious that we can use the `map` operation to transform the `Stream` of artists into a `Stream` of `String` names. Example 5-18 is a refactoring of this code to use streams and `map`.

Example 5-18. Using a forEach and a StringBuilder to pretty-print the names of artists

```
StringBuilder builder = new StringBuilder("[");
artists.stream()
       .map(Artist::getName)
       .forEach(name -> {
           if (builder.length() > 1)
               builder.append(", ");

           builder.append(name);
       });
builder.append("]");
String result = builder.toString();
```

This has made things a bit clearer in the sense that the mapping to names shows us what has been built up a bit more quickly. Unfortunately, there's still this very large `forEach` block that doesn't fit into our goal of writing code that is easy to understand by composing high-level operations.

Let's put aside our goal of building a custom collector for a moment and just think in terms of the existing operations that we have on streams. The operation that most closely matches what we're doing in terms of building up a `String` is the `reduce` operation. Refactoring Example 5-18 to use that results in Example 5-19.

Example 5-19. Using a reduce and a StringBuilder to pretty-print the names of artists

```
StringBuilder reduced =
    artists.stream()
           .map(Artist::getName)
           .reduce(new StringBuilder(), (builder, name) -> {
                   if (builder.length() > 0)
                       builder.append(", ");

                   builder.append(name);
                   return builder;
               }, (left, right) -> left.append(right));

reduced.insert(0, "[");
reduced.append("]");
String result = reduced.toString();
```

I had hoped that last refactor would help us make the code clearer. Unfortunately, it seems to be just as bad as before. Still, let's see what's going on. The `stream` and `map` calls are the same as in the previous example. Our `reduce` operation builds up the artist names, combined with `", "` delimiters. We start with an empty `StringBuilder`—the identity of the `reduce`. Our next lambda expression combines a name with a builder. The third argument to `reduce` takes two `StringBuilder` instances and combines them. Our final step is to add the prefix at the beginning and the suffix at the end.

For our next refactoring attempt, let's try and stick with reduction but hide the mess—I mean, abstract away the details—behind a class that we'll call a `StringCombiner`. Implementing this results in Example 5-20.

Example 5-20. Using a reduce and a custom StringCombiner to pretty-print the names of artists

```
StringCombiner combined =
        artists.stream()
               .map(Artist::getName)
               .reduce(new StringCombiner(", ", "[", "]"),
                       StringCombiner::add,
                       StringCombiner::merge);

String result = combined.toString();
```

Even though this looks quite different from the previous code example, it's actually doing the exact same thing under the hood. We're using `reduce` in order to combine names and delimiters into a `StringBuilder`. This time, though, the logic of adding elements is being delegated to the `StringCombiner.add` method and the logic of combining two different combiners is delegated to `StringCombiner.merge`. Let's take a look at these methods now, beginning with the `add` method in Example 5-21.

Example 5-21. The add method of a StringCombiner returns itself with a new element appended

```
public StringCombiner add(String element) {
    if (areAtStart()) {
        builder.append(prefix);
    } else {
        builder.append(delim);
    }
    builder.append(element);
    return this;
}
```

`add` is implemented by delegating operations to an underlying `StringBuilder` instance. If we're at the start of the combining operations, then we append our prefix; otherwise, we append the string that fits between our elements (the delimiter). We follow this up by appending the element. We return the `StringCombiner` object because this is the

value that we're pushing through our `reduce` operation. The merging code, provided in Example 5-22, delegates to appending operations on the `StringBuilder`.

Example 5-22. The merge method of a StringCombiner combines the results of both StringCombiners

```
public StringCombiner merge(StringCombiner other) {
    builder.append(other.builder);
    return this;
}
```

We're nearly done with the `reduce` phase of refactoring, but there's one small step remaining. We're going to inline the `toString` to the end of the method call chain so that our entire sequence is method-chained. This is simply a matter of lining up the `reduce` code so that it's ready to be converted into the Collector API (see Example 5-23).

Example 5-23. Using a reduce and delegating to our custom StringCombiner

```
String result =
    artists.stream()
            .map(Artist::getName)
            .reduce(new StringCombiner(", ", "[", "]"),
                    StringCombiner::add,
                    StringCombiner::merge)
            .toString();
```

At this stage, we have some code that looks vaguely sane, but it's quite hard to reuse this same combining operation in different parts of our code base. So we're going to refactor our `reduce` operation into a `Collector`, which we can use anywhere in our application. I've called our `Collector` the `StringCollector`. Let's refactor our code to use it in Example 5-24.

Example 5-24. Collecting strings using a custom StringCollector

```
String result =
    artists.stream()
            .map(Artist::getName)
            .collect(new StringCollector(", ", "[", "]"));
```

Now that we're delegating the whole of the `String`-joining behavior to a custom collector, our application code doesn't need to understand anything about the internals of `StringCollector`. It's just another `Collector` like any in the core framework.

We begin by implementing the `Collector` interface (Example 5-25). `Collector` is generic, so we need to determine a few types to interact with:

- The type of the element that we'll be collecting, a `String`
- Our accumulator type, `StringCombiner`, which you've already seen

- The result type, also a `String`

Example 5-25. How to define a collector over strings

```
public class StringCollector implements Collector<String, StringCombiner, String> {
```

A `Collector` is composed of four different components. First we have a `supplier`, which is a factory for making our container—in this case, a `StringCombiner`. The analogue to this is the first argument provided to the `reduce` operation, which was the initial value of the `reduce` (see Example 5-26).

Example 5-26. A supplier is a factory for making our container

```
public Supplier<StringCombiner> supplier() {
    return () -> new StringCombiner(delim, prefix, suffix);
}
```

Let's step through this in diagram form while we're walking through the code so that we can see how things fit together. Because collectors can be collected in parallel, we will show a collecting operation where two container objects (e.g., `StringCombiners`) are used in parallel.

Each of the four components of our `Collector` are functions, so we'll represent them as arrows. The values in our `Stream` are circles, and the final value we're producing will be an oval. At the start of the collect operation our supplier is used to create new container objects (see Figure 5-3).

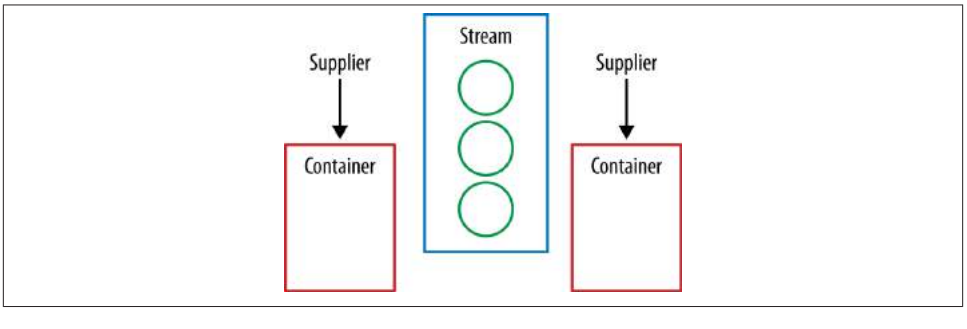

Figure 5-3. Supplier

Our collector's `accumulator` performs the same job as the second argument to `reduce`. It takes the current element and the result of the preceding operation and returns a new value. We've already implemented this logic in the `add` method of our `StringCombiner`, so we just refer to that (see Example 5-27).

Example 5-27. An accumulator is a function to fold the current element into the collector

```
public BiConsumer<StringCombiner, String> accumulator() {
    return StringCombiner::add;
}
```

Our accumulator is used to fold the stream's values into the container objects (Figure 5-4).

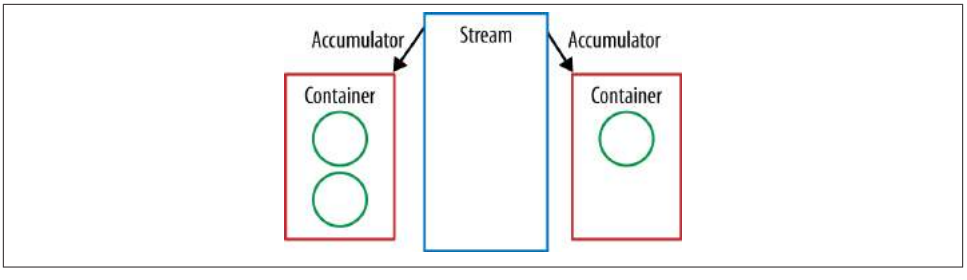

Figure 5-4. Accumulator

The `combine` method is an analogue of the third method of our `reduce` operation. If we have two containers, then we need to be able to merge them together. Again, we've already implemented this in a previous refactor step, so we just use the `StringCombiner.merge` method (Example 5-28).

Example 5-28. A combiner merges together two containers

```
public BinaryOperator<StringCombiner> combiner() {
    return StringCombiner::merge;
}
```

During the collect operation, our container objects are pairwise merged using the defined `combiner` until we have only one container at the end (Figure 5-5).

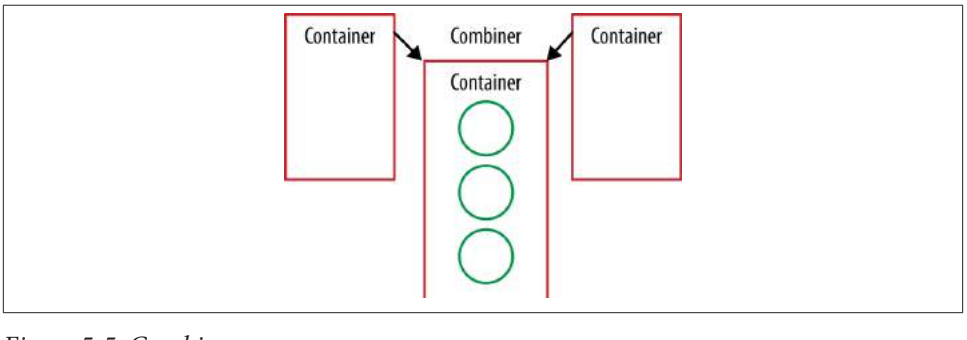

Figure 5-5. Combiner

You might remember that the last step in our refactoring process, before we got to collectors, was to put the `toString` method inline at the end of the method chain. This converted our `StringCombiner` into the `String` that we really wanted (Figure 5-6).

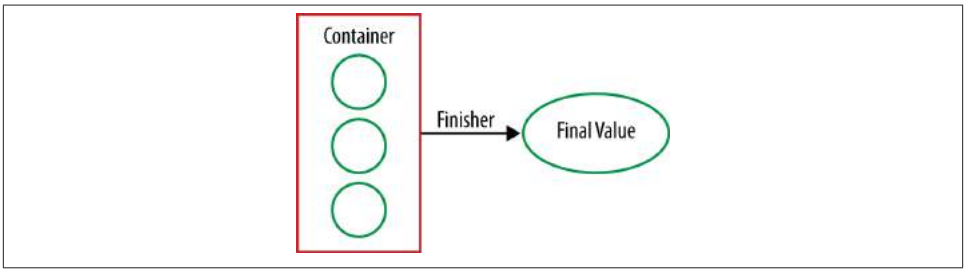

Figure 5-6. Finisher

Our collector's `finisher` method performs the same purpose. We've already folded our mutable container over a `Stream` of values, but it's not quite the final value that we want. The `finisher` gets called here, once, in order to make that conversion. This is especially useful if we want to create an immutable final value, such as a `String`, but our container is mutable.

In order to implement the `finisher` for this operation, we just delegate to the `toString` method that we've already written (Example 5-29).

Example 5-29. A finisher produces the final value returned by the collect operation

```
public Function<StringCombiner, String> finisher() {
    return StringCombiner::toString;
}
```

We create our final value from the one remaining container.

There's one aspect of collectors that I haven't described so far: *characteristics*. A characteristic is a `Set` of objects that describes the `Collector`, allowing the framework to perform certain optimizations. It's defined through a `characteristics` method.

At this juncture, it's worth reminding ourselves that this code has been written as an educational exercise and differs a little bit from the internal implementation of the `joining` collector. You may also be thinking that the `StringCombiner` class is looking quite useful. Don't worry—you don't need to write that either! Java 8 contains a `java.util.StringJoiner` class that performs a similar role and has a similar API.

The main goals of going through this exercise are not only to show how custom collectors work, but also to allow you to write your own collector. This is especially useful if you have a domain class that you want to build up from an operation on a collection and none of the standard collectors will build it for you.

In the case of our `StringCollector`, the container that we were using to collect values was different from the final value that we were trying to create (a `String`). This is especially common if you're trying to collect immutable values rather than mutable ones, because otherwise each step of the collection operation would have to create a new value.

It's entirely possible for the final value that you're collecting to be the same as the container you've been folding your values into all along. In fact, this is what happens when the final value that you're collecting is a `Collection`, such as with the `toList` collector.

In this case, your `finisher` method needs to do nothing to its container object. More formally, we can say that the `finisher` method is the `identity` function: it returns the value passed as an argument. If this is the case, then your `Collector` will exhibit the `IDENTITY_FINISH` characteristic and should declare it using the `characteristics` method.

Reduction as a Collector

As you've just seen, custom collectors aren't that hard to write, but if you're thinking about writing one in order to collect into a domain class it is worth examining the alternatives. The most obvious is to build one or more collection objects and then pass them into the constructor of your domain class. This is really simple and suitable if your domain class is a composite containing different collections.

Of course, if your domain class isn't just a composite and needs to perform some calculation based on the existing data, then that isn't a suitable route. Even in this situation, though, you don't necessarily need to build up a custom collector. You can use the `reducing` collector, which gives us a generic implementation of the reduction operation over streams. Example 5-30 shows how we might write our `String`-processing example using the `reducing` collector.

Example 5-30. Reducing is a convenient way of making custom collectors

```
String result =
        artists.stream()
                .map(Artist::getName)
                .collect(Collectors.reducing(
                    new StringCombiner(", ", "[", "]"),
                    name -> new StringCombiner(", ", "[", "]").add(name),
                    StringCombiner::merge))
                .toString();
```

This is very similar to the `reduce`-based implementation I covered in Example 5-20, which is what you might expect given the name. The key difference is the second argument to `Collectors.reducing`; we are creating a dedicated `StringCombiner` for each element in the stream. If you are shocked or disgusted at this, you should be! This is highly inefficient and one of the reasons why I chose to write a custom collector.

Collection Niceties

The introduction of lambda expressions has also enabled other collection methods to be introduced. Let's have a look at some useful changes that have been made to `Map`.

A common requirement when building up a `Map` is to compute a value for a given key. A classic example of this is when implementing a cache. The traditional idiom is to try and retrieve a value from the `Map` and then create it, if it's not already there.

If we defined our cache as `Map<String, Artist> artistCache` and were wanting to look up artists using an expensive database operation, we might write something like Example 5-31.

Example 5-31. Caching a value using an explicit null check

```
public Artist getArtist(String name) {
    Artist artist = artistCache.get(name);
    if (artist == null) {
        artist = readArtistFromDB(name);
        artistCache.put(name, artist);
    }
    return artist;
}
```

Java 8 introduces a new `computeIfAbsent` method that takes a lambda to compute the new value if it doesn't already exist. So, we can rewrite the previous block of code into Example 5-32.

Example 5-32. Caching a value using computeIfAbsent

```
public Artist getArtist(String name) {
    return artistCache.computeIfAbsent(name, this::readArtistFromDB);
}
```

You may want variants of this code that don't perform computation only if the value is absent; the new `compute` and `computeIfPresent` methods on the `Map` interface are useful for these cases.

At some point in your career, you might have tried to iterate over a `Map`. Historically, the approach was to use the `values` method to get a `Set` of entries and then iterate over them. This tended to result in fairly hard-to-read code. Example 5-33 shows an approach from earlier in the chapter of creating a new `Map` counting the number of albums associated with each artist.

Example 5-33. An ugly way to iterate over all entries of a Map

```
Map<Artist, Integer>  countOfAlbums = new HashMap<>();
for(Map.Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
    Artist artist = entry.getKey();
    List<Album> albums = entry.getValue();
```

```
    countOfAlbums.put(artist, albums.size());
}
```

Thankfully, a new `forEach` method has been introduced that takes a `BiConsumer` (two values enter, nothing leaves) and produces easier-to-read code through internal iteration, which I introduced in "From External Iteration to Internal Iteration" on page 17. An equivalent code sample is shown in Example 5-34.

Example 5-34. Using internal iteration over all entries of a Map

```
Map<Artist, Integer>  countOfAlbums = new HashMap<>();
albumsByArtist.forEach((artist, albums) -> {
    countOfAlbums.put(artist, albums.size());
});
```

Key Points

- Method references are a lightweight syntax for referring to methods and look like this: `ClassName::methodName`.
- Collectors let us compute the final values of streams and are the mutable analogue of the `reduce` method.
- Java 8 provides out-of-the-box support for collecting into many collection types and the ability to build custom collectors.

Exercises

1. *Method references.* Take a look back at the examples in Chapter 3 and try rewriting the following using method references:
   a. The `map` to uppercase
   b. The implementation of `count` using `reduce`
   c. The `flatMap` approach to concatenating lists
2. *Collectors.*
   a. Find the artist with the longest name. You should implement this using a `Collector` and the `reduce` higher-order function from Chapter 3. Then compare the differences in your implementation: which was easier to write and which was easier to read? The following example should return `"Stuart Sutcliffe"`:

```
Stream<String> names = Stream.of("John Lennon", "Paul McCartney",
    "George Harrison", "Ringo Starr", "Pete Best", "Stuart Sutcliffe");
```

b. Given a `Stream` where each element is a word, count the number of times each word appears. So, if you were given the following input, you would return a `Map` of `[John → 3, Paul → 2, George → 1]`:

```
Stream<String> names = Stream.of("John", "Paul", "George", "John",
                                 "Paul", "John");
```

c. Implement `Collectors.groupingBy` as a custom collector. You don't need to provide a downstream collector, so just implementing the simplest variant is fine. If you look at the JDK source code, you're cheating! Hint: you might want to start with `public class GroupingBy<T, K> implements Collector<T, Map<K, List<T>>, Map<K, List<T>>>`. This is an advanced exercise, so you might want to attempt it last.

3. *Map enhancements.*

   Efficiently calculate a Fibonacci sequence using just the `computeIfAbsent` method on a `Map`. By "efficiently," I mean that you don't repeatedly recalculate the Fibonacci sequence of smaller numbers.

CHAPTER 6
Data Parallelism

I've previously made a lot of references to the idea that it's easier to write parallel code in Java 8. This is because we can use lambda expressions in combination with the streams library, introduced in Chapter 3, to say what we want our program to do, regardless of whether it's sequential or parallel. I know that sounds a lot like what you've been doing in Java for years, but there's a difference between saying what you want to compute and saying how to compute it.

The big shift between external and internal iteration (also discussed in Chapter 3) did make it easier to write simple and clean code, but here's the other big benefit: now we don't have to manually control the iteration. It doesn't need to be performed sequentially. We express the what and, by changing a single method call, we can get a library to figure out the how.

The changes to your code are surprisingly unobtrusive, so the majority of this chapter won't be talking about how your code changes. Instead, I'll explain why you might want to go parallel and when you'll get performance improvements. It's also worth noting that this chapter isn't a general text on performance in Java; we'll just be looking at the easy wins provided in Java 8.

Parallelism Versus Concurrency

After a quick scan over the table of contents of this book, you might have noticed this chapter with the word *parallelism* in the title and also Chapter 9, which has *concurrency* in the title. Don't worry—I haven't repeated the same material in an attempt to justify charging you more for this book! Concurrency and parallelism are different things that can be leveraged to achieve different aims.

Concurrency arises when two tasks are making progress at overlapping time periods. Parallelism arises when two tasks are happening at literally the same time, such as on a multicore CPU. If a program is undertaking two tasks and they are being given small

slices of a single CPU core's time, then it is exhibiting concurrency but not parallelism. This difference is shown in Figure 6-1.

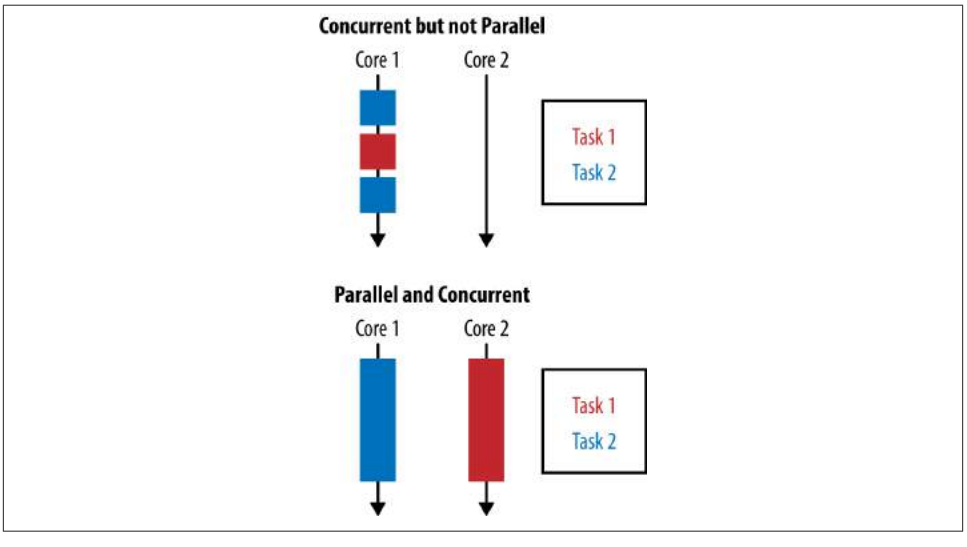

Figure 6-1. Comparison of concurrency and parallelism

The goal of parallelism is to reduce the runtime of a specific task by breaking it down into smaller components and performing them in parallel. This doesn't mean that you won't do as much work as you would if you were running them sequentially—you are just getting more horses to pull the same cart for a shorter time period. In fact, it's usually the case that running a task in parallel requires more work to be done by the CPU than running it sequentially would.

In this chapter, we're looking at a very specific form of parallelism called *data parallelism*. In data parallelism, we achieve parallelism by splitting up the data to be operated on and assigning a single processing unit to each chunk of data. If we're to extend our horses-pulling-carts analogy, it would be like taking half of the goods inside our cart and putting them into another cart for another horse to pull, with both horses taking an identical route to the destination.

Data parallelism works really well when you want to perform the same operation on a lot of data. The problem needs be decomposed in a way that will work on subsections of the data, and then the answers from each subsection can be composed at the end.

Data parallelism is often contrasted with *task parallelism*, in which each individual thread of execution can be doing a totally different task. Probably the most commonly encountered task parallelism is a Java EE application container. Each thread not only

can be dealing with processing a different user, but also could be performing different tasks for a user, such as logging in or adding an item to a shopping cart.

Why Is Parallelism Important?

Historically, we could all rely on the clock frequency of a CPU getting faster over time. Intel's 8086 processor, introduced in 1979, started at a 5 MHz clock rate, and by the time the Pentium chip was introduced in 1993 speeds had reached 60 MHz. Improved sequential performance continued through the early 2000s.

Over the last decade, however, mainstream chip manufacturers have been moving increasingly toward heavily multicore processors. At the time of writing, it's not uncommon for servers to be shipping with 32 or 64 cores spread over several physical processing units. This trend shows no sign of abating soon.

This influences the design of software. Instead of being able to rely on improved CPU clock speeds to increase the computational capacity of existing code, we need to be able to take advantage of modern CPU architectures. The only way to do this is by writing parallel programs.

I appreciate you've probably heard this message before. In fact, it's one that's been blasted out by many conference speakers, book authors, and consultants over the years. The implications of *Amdahl's Law* were what really made me take note of the importance of parallelism.

Amdahl's Law is a simple rule that predicts the theoretical maximum speedup of a program on a machine with multiple cores. If we take a program that is entirely serial and parallelize only half of it, then the maximum speedup possible, regardless of how many cores we throw at the problem, is 2×. Given a large number of cores—and we're already into that territory—the execution time of a problem is going to be dominated by the serial part of that problem.

When you start to think of performance in these terms, optimizing any job that is bound by computational work rapidly becomes a matter of ensuring that it effectively utilizes the available hardware. Of course, not every job is bound by computational work, but in this chapter we'll be focusing on that kind of problem.

Parallel Stream Operations

Making an operation execute in parallel using the streams library is a matter of changing a single method call. If you already have a `Stream` object, then you can call its `parallel` method in order to make it parallel. If you're creating a `Stream` from a `Collection`, you can call the `parallelStream` method in order to create a parallel stream from the get-go.

Let's look at a simple example in order to make things concrete. Example 6-1 calculates the total length of a sequence of albums. It transforms each album into its component tracks, then gets into the length of each track, and then sums them.

Example 6-1. Serial summing of album track lengths

```
public int serialArraySum() {
    return albums.stream()
                 .flatMap(Album::getTracks)
                 .mapToInt(Track::getLength)
                 .sum();
}
```

We go parallel by making the call to `parallelStream`, as shown in Example 6-2; all the rest of the code is identical. Going parallel *just works*.

Example 6-2. Parallel summing of album track lengths

```
public int parallelArraySum() {
    return albums.parallelStream()
                 .flatMap(Album::getTracks)
                 .mapToInt(Track::getLength)
                 .sum();
}
```

I know the immediate instinct upon hearing this is to go out and replace every call to `stream` with a call to `parallelStream` because it's so easy. Hold your horses for a moment! Obviously it's important to make good use of parallelism in order to get the most from your hardware, but the kind of data parallelism we get from the streams library is only one form.

The question we really want to ask ourselves is whether it's faster to run our `Stream`-based code sequentially or in parallel, and that's not a question with an easy answer. If we look back at the previous example, where we figure out the total running time of a list of albums, depending upon the circumstances we can make the sequential or parallel versions faster.

When benchmarking the code in Examples 6-1 and 6-2 on a 4-core machine with 10 albums, the sequential code was 8× faster. Upon expanding the number of albums to 100, they were both equally fast, and by the time we hit 10,000 albums, the parallel code was 2.5× faster.

Any specific benchmark figures in this chapter are listed only to make a point. If you try to replicate these results on your hardware, you may get drastically different outcomes.

The size of the input stream isn't the only factor to think about when deciding whether there's a parallel speedup. It's possible to get varying performance numbers based upon how you wrote your code and how many cores are available. We'll look at this in a bit more detail in "Performance" on page 89, but first let's look at a more complex example.

Simulations

The kinds of problems that parallel stream libraries excel at are those that involve simple operations processing a lot of data, such as simulations. In this section, we'll be building a simple simulation to understand dice throws, but the same ideas and approach can be used on larger and more realistic problems.

The kind of simulation we'll be looking at here is a *Monte Carlo* simulation. Monte Carlo simulations work by running the same simulation many times over with different random seeds on every run. The results of each run are recorded and aggregated in order to build up a comprehensive simulation. They have many uses in engineering, finance, and scientific computing.

If we throw a fair die twice and add up the number of dots on the winning side, we'll get a number between 2 and 12. This must be at least 2 because the fewest number of dots on each side is 1 and there are two dice. The maximum score is 12, as the highest number you can score on each die is 6. We want to try and figure out what the probability of each number between 2 and 12 is.

One approach to solving this problem is to add up all the different combinations of dice rolls that can get us each value. For example, the only way we can get 2 is by rolling 1 and then 1 again. There are 36 different possible combinations, so the probability of the two sides adding up to 2 is 1 in 36, or 1/36.

Another way of working it out is to simulate rolling two dice using random numbers between 1 and 6, adding up the number of times that each result was picked, and dividing by the number of rolls. This is actually a really simple Monte Carlo simulation. The more times we simulate rolling the dice, the more closely we approximate the actual result—so we really want to do it a lot.

Example 6-3 shows how we can implement the Monte Carlo approach using the streams library. `N` represents the number of simulations we'll be running, and at ❶ we use the `IntStream` range function to create a stream of size `N`. At ❷ we call the `parallel` method in order to use the parallel version of the streams framework. The `twoDiceThrows` function simulates throwing two dice and returns the sum of their results. We use the `mapToObj` method in ❸ in order to use this function on our data stream.

Example 6-3. Parallel Monte Carlo simulation of dice rolling

```
public Map<Integer, Double> parallelDiceRolls() {
    double fraction = 1.0 / N;
```

```
    return IntStream.range(0, N)                          ❶
                    .parallel()                           ❷
                    .mapToObj(twoDiceThrows())            ❸
                    .collect(groupingBy(side -> side,     ❹
                        summingDouble(n -> fraction)));   ❺
}
```

At ❹ we have a `Stream` of all the simulation results we need to combine. We use the `groupingBy` collector, introduced in the previous chapter, in order to aggregate all results that are equal. I said we were going to count the number of times each number occured and divide by `N`. In the streams framework, it's actually easier to map numbers to `1/N` and add them, which is exactly the same. This is accomplished in ❺ through the `summingDouble` function. The `Map<Integer, Double>` that gets returned at the end maps each sum of sides thrown to its probability.

I'll admit it's not totally trivial code, but implementing a parallel Monte Carlo simulation in five lines of code is pretty neat. Importantly, because the more simulations we run, the more closey we approximate the real answer, we've got a real incentive to run a lot of simulations. This is also a good use for parallelism as it's an implementation that gets good parallel speedup.

I won't go through the implementation details, but for comparison Example 6-4 lists the same parallel Monte Carlo simulation implemented by hand. The majority of the code implementation deals with spawning, scheduling, and awaiting the completion of jobs within a thread pool. None of these issues needs to be directly addressed when using the parallel streams library.

Example 6-4. Simulating dice rolls by manually implementing threading

```
public class ManualDiceRolls {

    private static final int N = 100000000;

    private final double fraction;
    private final Map<Integer, Double> results;
    private final int numberOfThreads;
    private final ExecutorService executor;
    private final int workPerThread;

    public static void main(String[] args) {
        ManualDiceRolls roles = new ManualDiceRolls();
        roles.simulateDiceRoles();
    }

    public ManualDiceRolls() {
        fraction = 1.0 / N;
        results = new ConcurrentHashMap<>();
        numberOfThreads = Runtime.getRuntime().availableProcessors();
        executor = Executors.newFixedThreadPool(numberOfThreads);
```

```
        workPerThread = N / numberOfThreads;
    }

    public void simulateDiceRoles() {
        List<Future<?>> futures = submitJobs();
        awaitCompletion(futures);
        printResults();
    }

    private void printResults() {
        results.entrySet()
               .forEach(System.out::println);
    }

    private List<Future<?>> submitJobs() {
        List<Future<?>> futures = new ArrayList<>();
        for (int i = 0; i < numberOfThreads; i++) {
            futures.add(executor.submit(makeJob()));
        }
        return futures;
    }

    private Runnable makeJob() {
        return () -> {
            ThreadLocalRandom random = ThreadLocalRandom.current();
            for (int i = 0; i < workPerThread; i++) {
                int entry = twoDiceThrows(random);
                accumulateResult(entry);
            }
        };
    }

    private void accumulateResult(int entry) {
        results.compute(entry, (key, previous) ->
            previous == null ? fraction
                             : previous + fraction
        );
    }

    private int twoDiceThrows(ThreadLocalRandom random) {
        int firstThrow = random.nextInt(1, 7);
        int secondThrow = random.nextInt(1, 7);
        return firstThrow + secondThrow;
    }

    private void awaitCompletion(List<Future<?>> futures) {
        futures.forEach((future) -> {
            try {
                future.get();
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
```

```
        });
        executor.shutdown();
    }

}
```

Caveats

I said earlier that using parallel streams "just works," but that's being a little cheeky. You can run existing code in parallel with little modification, but only if you've written idiomatic code. There are a few rules and restrictions that need to be obeyed in order to make optimal use of the parallel streams framework.

Previously, when calling `reduce` our initial element could be any value, but for this operation to work correctly in parallel, it needs to be the *identity* value of the combining function. The identity value leaves all other elements the same when reduced with them. For example, if we're summing elements with our `reduce` operation, the combining function is `(acc, element) -> acc + element`. The initial element must be `0`, because any number `x` added to `0` returns `x`.

The other caveat specific to `reduce` is that the combining function must be *associative*. This means that the order in which the combining function is applied doesn't matter as long the values of the sequence aren't changed. Confused? Don't worry! Take a look at Example 6-5, which shows how we can rearrange the order in which we apply + and * to a sequence of values and get the same result.

*Example 6-5. + and * are associative*

```
(4 + 2) + 1 = 4 + (2 + 1) = 7
(4 * 2) * 1 = 4 * (2 * 1) = 8
```

One thing to avoid is trying to hold locks. The streams framework deals with any necessary synchronization itself, so there's no need to lock your data structures. If you do try to hold locks on any data structure that the streams library is using, such as the source collection of an operation, you're likely to run into trouble.

I explained earlier that you could convert any existing `Stream` to be a parallel stream using the `parallel` method call. If you've been looking at the API itself while reading the book, you may have noticed a `sequential` method as well. When a stream pipeline is evaluated, there is no mixed mode: the orientation is either parallel or sequential. If a pipeline has calls to both `parallel` and `sequential`, the last call wins.

Performance

I briefly mentioned before that there were a number of factors that influenced whether parallel streams were faster or slower than sequential streams; let's take a look at those factors now. Understanding what works well and what doesn't will help you to make an informed decision about how and when to use parallel streams. There are five important factors that influence parallel streams performance that we'll be looking at:

Data size
: There is a difference in the efficiency of the parallel speedup due to the size of the input data. There's an overhead to decomposing the problem to be executed in parallel and merging the results. This makes it worth doing only when there's enough data that execution of a streams pipeline takes a while. We explored this back in "Parallel Stream Operations" on page 83.

Source data structure
: Each pipeline of operations operates on some initial data source; this is usually a collection. It's easier to split out subsections of different data sources, and this cost affects how much parallel speedup you can get when executing your pipeline.

Packing
: Primitives are faster to operate on than boxed values.

Number of cores
: The extreme case here is that you have only a single core available to operate upon, so it's not worth going parallel. Obviously, the more cores you have access to, the greater your potential speedup is. In practice, what counts isn't just the number of cores allocated to your machine; it's the number of cores that are available for your machine to use at runtime. This means factors such as other processes executing simultaneously or thread affinity (forcing threads to execute on certain cores or CPUs) will affect performance.

Cost per element
: Like data size, this is part of the battle between time spent executing in parallel and overhead of decomposition and merging. The more time spent operating on each element in the stream, the better performance you'll get from going parallel.

When using the parallel streams framework, it can be helpful to understand how problems are decomposed and merged. This gives us a good insight into what is going on under the hood without having to understand all the details of the framework.

Let's take a look at how a concrete example is decomposed and merged. Example 6-6 shows some code that performs parallel integer addition.

Example 6-6. Parallel integer addition

```
private int addIntegers(List<Integer> values) {
    return values.parallelStream()
                 .mapToInt(i -> i)
                 .sum();
}
```

Under the hood, parallel streams back onto the fork/join framework. The *fork* stage recursively splits up a problem. Then each chunk is operated upon in parallel. Finally, the *join* stage merges the results back together.

Figure 6-2 shows how this might apply to Example 6-6.

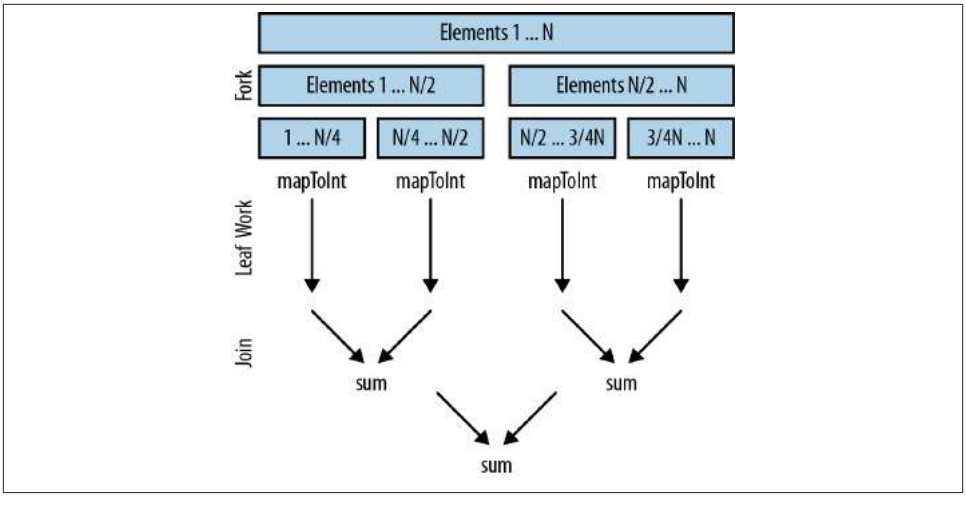

Figure 6-2. Decomposing and merging using fork/join

Let's assume that the streams framework is splitting up our work to operate in parallel on a four-core machine:

1. Our data source is decomposed into four chunks of elements.
2. We perform leaf computation work in parallel on each thread in Example 6-6. This involves mapping each `Integer` to an `int` and also summing a quarter of the values in each thread. Ideally, we want to spend as much of our time as possible in leaf computation work because it's the perfect case for parallelism.
3. We merge the results. In Example 6-6 this is just a `sum` operation, but it might involve any kind of `reduce`, `collect`, or terminal operation.

Given the way problems are decomposed, the nature of the initial source is extremely important in influencing the performance of this decomposition. Intuitively, the ease

with which we can repeatedly split a data structure in half corresponds to how fast it can be operated upon. Splitting in half also means that the values to be operated upon need to split equally.

We can split up common data sources from the core library into three main groups by performance characteristics:

The good
: An `ArrayList`, an array, or the `IntStream.range` constructor. These data sources all support random access, which means they can be split up arbitrarily with ease.

The okay
: The `HashSet` and `TreeSet`. You can't easily decompose these with perfect amounts of balance, but most of the time it's possible to do so.

The bad
: Some data structures just don't split well; for example, they may take $O(N)$ time to decompose. Examples here include a `LinkedList`, which is computationally hard to split in half. Also, `Streams.iterate` and `BufferedReader.lines` have unknown length at the beginning, so it's pretty hard to estimate when to split these sources.

The influence of the initial data structure can be huge. To take an extreme example, benchmarking a parallel sum over 10,000 integers revealed an `ArrayList` to be 10 times faster than a `LinkedList`. This isn't to say that your business logic will exhibit the same performance characteristics, but it does demonstrate how influential these things can be. It's also far more likely that data structures such as a `LinkedList` that have poor decompositions will also be slower when run in parallel.

Ideally, once the streams framework has decomposed the problem into smaller chunks, we'll be able to operate on each chunk in its own thread, with no further communication or contention between threads. Unfortunately, reality can get the way of the ideal at times!

When we're talking about the kinds of operations in our stream pipeline that let us operate on chunks individually, we can differentiate between two types of stream operations: *stateless* and *stateful*. Stateless operations need to maintain no concept of state over the whole operation; stateful operations have the overhead and constraint of maintaining state.

If you can get away with using stateless operations, then you will get better parallel performance. Examples of stateless operations include `map`, `filter`, and `flatMap`; `sorted`, `distinct`, and `limit` are stateful.

Performance-test your own code. The advice in this section offers rules of thumb about what performance characteristics should be investigated, but nothing beats measuring and profiling.

Parallel Array Operations

Java 8 includes a couple of other parallel array operations that utilize lambda expressions outside of the streams framework. Like the operations on the streams framework, these are data parallel operations. Let's look at how we can use these operations to solve problems that are hard to do in the streams framework.

These operations are all located on the utility class `Arrays`, which also contains a bunch of other useful array-related functionality from previous Java versions. There is a summary in Table 6-1.

Table 6-1. Parallel operations on arrays

Name	Operation
`parallelPrefix`	Calculates running totals of the values of an array given an arbitrary function
`parallelSetAll`	Updates the values in an array using a lambda expression
`parallelSort`	Sorts elements in parallel

You may have written code similar to Example 6-7 before, where you initialize an array using a `for` loop. In this case, we initialize every element to its index in the array.

Example 6-7. Initializing an array using a for loop

```
public static double[] imperativeInitilize(int size) {
    double[] values = new double[size];
    for(int i = 0; i < values.length;i++) {
        values[i] = i;
    }
    return values;
}
```

We can use the `parallelSetAll` method in order to do this easily in parallel. An example of this code is shown in Example 6-8. We provide an array to operate on and a lambda expression, which calculates the value given the index. In our example they are the same value. One thing to note about these methods is that they alter the array that is passed into the operation, rather than creating a new copy.

Example 6-8. Initializing an array using a parallel array operation

```
public static double[] parallelInitialize(int size) {
    double[] values = new double[size];
    Arrays.parallelSetAll(values, i -> i);
```

```
    return values;
}
```

The `parallelPrefix` operation, on the other hand, is much more useful for performing accumulation-type calculations over time series of data. It mutates an array, replacing each element with the *sum* of that element and its predecessors. I use the term "sum" loosely—it doesn't need to be addition; it could be any `BinaryOperator`.

An example operation that can be calculated by prefix sums is a simple moving average. This takes a rolling window over a time series and produces an average for each instance of that window. For example, if our series of input data is `0, 1, 2, 3, 4, 3.5`, then the simple moving average of size 3 is `1, 2, 3, 3.5`. Example 6-9 shows how we can use a prefix sum in order to calculate a moving average.

Example 6-9. Calculating a simple moving average

```
public static double[] simpleMovingAverage(double[] values, int n) {
    double[] sums = Arrays.copyOf(values, values.length);  ❶
    Arrays.parallelPrefix(sums, Double::sum);  ❷
    int start = n - 1;
    return IntStream.range(start, sums.length)  ❸
                    .mapToDouble(i -> {
                        double prefix = i == start ? 0 : sums[i - n];
                        return (sums[i] - prefix) / n;  ❹
                    })
                    .toArray();  ❺
}
```

It's quite complex, so I'll go through how this works in a few steps. The input parameter `n` is the size of the time window we're calculating our moving average over. At ❶ we take a copy of our input data. Because our prefix calculation is a mutating operation, we do this to avoid altering the original source.

In ❷ we apply the prefix operation, adding up values in the process. So now our `sums` variable holds the running total of the sums so far. For example, given the input `0, 1, 2, 3, 4, 3.5`, it would hold `0.0, 1.0, 3.0, 6.0, 10.0, 13.5`.

Now that we have the complete running totals, we can find the sum over the time window by subtracting the running total at the beginning of the time window. The average is this divided by `n`. We can do this calculation using the existing streams library, so let's use it! We kick off the stream in ❸ by using `Intstream.range` to get a stream ranging over the indices of the values we want.

At ❹ we subtract away the running total at the start and then do the division in order to get the average. It's worth noting that there's an edge case for the running total at element n – 1, where there is no running total to subtract to begin with. Finally, at ❺, we convert the `Stream` back to an array.

Key Points

- Data parallelism is a way to split up work to be done on many cores at the same time.
- If we use the streams framework to write our code, we can utilize data parallelism by calling the `parallel` or `parallelStream` methods.
- The five main factors influencing performance are the data size, the source data structure, whether the values are packed, the number of available cores, and how much processing time is spent on each element.

Exercises

1. The code in Example 6-10 sequentially sums the squares of numbers in a `Stream`. Make it run in parallel using streams.

   Example 6-10. Sequentially summing the squares of numbers in a list

   ```
   public static int sequentialSumOfSquares(IntStream range) {
       return range.map(x -> x * x)
                   .sum();
   }
   ```

2. The code in Example 6-11 multiplies every number in a list together and multiplies the result by 5. This works fine sequentially, but has a bug when running in parallel. Make the code run in parallel using streams and fix the bug.

   Example 6-11. A buggy way of multiplying every number in a list together and multiplying the result by 5

   ```
   public static int multiplyThrough(List<Integer> linkedListOfNumbers) {
       return linkedListOfNumbers.stream()
                   .reduce(5, (acc, x) -> x * acc);
   }
   ```

3. The code in Example 6-12 also calculates the sum of the squares of numbers in a list. You should try to improve the performance of this code *without* degrading its quality. I'm only looking for you to make a couple of simple changes.

   Example 6-12. Slow implementation of summing the squares of numbers in a list

   ```
   public int slowSumOfSquares() {
       return linkedListOfNumbers.parallelStream()
                                 .map(x -> x * x)
                                 .reduce(0, (acc, x) -> acc + x);
   }
   ```

Make sure to run the benchmark code multiple times when timing. The sample code provided on GitHub comes with a benchmark harness that you can use.

CHAPTER 7

Testing, Debugging, and Refactoring

The rising popularity of techniques such as refactoring, test-driven development (TDD), and continuous integration (CI) mean that if we're going to use lambda expressions in our day-to-day programming, we need to understand how to test code using them and written with them.

A wealth of material has been written on how to test and debug computer programs, and this chapter isn't going to revisit all that material. If you're interested in learning how to do TDD properly, I highly recommend the books *Test-Driven Development* by Kent Beck and *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (both from Addison-Wesley).

I am going to cover techniques specific to using lambda expressions in your code, and when you might not want to (directly) use lambda expressions at all. I'll also talk about some appropriate techniques for debugging programs that heavily use lambda expressions and streams.

We're first going to look at some examples of how to refactor an existing code base into using lambda expressions. I've talked a bit already about how to do local refactoring operations, such as replacing a `for` loop with a stream operation. Here we'll take a more in-depth look at how non-collection code can be improved.

Lambda Refactoring Candidates

The process of refactoring code to take advantage of lambdas has been given the snazzy name *point lambdafication* (pronounced *lambda-fi-cation*, practitioners of this process being "lamb-di-fiers" or "responsible developers"). It's a process that has happened within the Java core libraries for Java 8. When you're choosing how to model the internal design of your application, it's also really worth considering which API methods to expose in this way.

There are a few key heuristics that can help you out when identifying an appropriate place to lambdify your application or library code. Each of these can be considered a localized antipattern or code smell that you're fixing through point lambdification.

In, Out, In, Out, Shake It All About

In Example 7-1, I've repeated our example code from Chapter 4 about logging statements. You'll see that it's pulling out the Boolean value from `isDebugEnabled` only to check it and then call a method on the `Logger`. If you find that your code is repeatedly querying and operating on an object only to push a value back into that object at the end, then that code belongs in the class of the object that you're modifying.

Example 7-1. A logger using isDebugEnabled to avoid performance overhead

```
Logger logger = new Logger();
if (logger.isDebugEnabled()) {
    logger.debug("Look at this: " + expensiveOperation());
}
```

Logging is a good example of where this has historically been difficult to achieve, because in different locations you're trying to provide different behavior. In this case, the behavior is building up a message string that will differ depending upon where in your program you're logging and what information you're trying to log.

This antipattern can be easily solved by passing in code as data. Instead of querying an object and then setting a value on it, you can pass in a lambda expression that represents the relevant behavior by computing a value. I've also repeated the code solution in Example 7-2, as a reminder. The lambda expression gets called if we're at a debug level and the logic for checking this call remains inside of the `Logger` itself.

Example 7-2. Using lambda expressions to simplify logging code

```
Logger logger = new Logger();
logger.debug(() -> "Look at this: " + expensiveOperation());
```

Logging is also a demonstration of using lambda expressions to do better object-oriented programming (OOP). A key OOP concept is to encapsulate local state, such as the level of the logger. This isn't normally encapsulated very well, as `isDebugEnabled` exposes its state. If you use the lambda-based approach, then the code outside of the logger doesn't need to check the level at all.

The Lonely Override

In this code smell, you subclass solely to override a single method. The `ThreadLocal` class is a good example of this. `ThreadLocal` allows us to create a factory that generates at most one value per thread. This is an easy way of ensuring that a thread-unsafe class can be safely used in a concurrent environment. For example, if we need to look up an

artist from the database but want to do it once per thread, then we might write something like the code in Example 7-3.

Example 7-3. Looking up an artist from the database

```
ThreadLocal<Album> thisAlbum = new ThreadLocal<Album> () {
    @Override protected Album initialValue() {
        return database.lookupCurrentAlbum();
    }
};
```

In Java 8 we can use the factory method `withInitial` and pass in a `Supplier` instance that deals with the creation, as shown in Example 7-4.

Example 7-4. Using the factory method

```
ThreadLocal<Album> thisAlbum
    = ThreadLocal.withInitial(() -> database.lookupCurrentAlbum());
```

There are a few reasons why the second example would be considered preferable to the first. For a start, any existing instance of `Supplier<Album>` can be used here without needing to be repackaged for this specific case, so it encourages reuse and composition.

It's also shorter to write, which is an advantage if and only if all other things are equal. More important, it's shorter because it's a lot cleaner: when reading the code, the signal-to-noise ratio is lower. This means you spend more time solving the actual problem at hand and less time dealing with subclassing boilerplate. It also has the advantage that it's one fewer class that your JVM has to load.

It's also a lot clearer to anyone who tries to read the code what its intent is. If you try to read out loud the words in the second example, you can easily hear what it's saying. You definitely can't say this of the first example.

Interestingly, this wasn't an antipattern previously to Java 8—it was the idiomatic way of writing this code, in the same way that using anonymous inner classes to pass around behavior wasn't an antipattern, just the only way of expressing what you wanted in Java code. As the language evolves, so do the idioms that you use when programming.

Behavioral Write Everything Twice

Write Everything Twice (WET) is the opposite of the well-known *Don't Repeat Yourself* (DRY) pattern. This code smell crops up in situations where your code ends up in repetitive boilerplate that produces more code that needs to be tested, is harder to refactor, and is brittle to change.

Not all WET situations are suitable candidates for point lambdification. In some situations, couple duplication can be the only alternative to having an overly closely coupled system. There's a good heuristic for situations where WET suggests it's time to add some

point lambdification into your application. Try adding lambdas where you want to perform a similar overall pattern but have a different behavior from one variant to another.

Let's look at a more concrete example. On top of our music domain, I've decided to add a simple `Order` class that calculates useful properties about some albums that a user wants to buy. We're going to count the number of musicians, number of tracks, and running time of our `Order`. If we were using imperative Java, we would write some code like Example 7-5.

Example 7-5. An imperative implementation of our Order class

```
public long countRunningTime() {
    long count = 0;
    for (Album album : albums) {
        for (Track track : album.getTrackList()) {
            count += track.getLength();
        }
    }
    return count;
}

public long countMusicians() {
    long count = 0;
    for (Album album : albums) {
        count += album.getMusicianList().size();
    }
    return count;
}

public long countTracks() {
    long count = 0;
    for (Album album : albums) {
        count += album.getTrackList().size();
    }
    return count;
}
```

In each case, we've got the boilerplate code of adding some code for each album to the total—for example, the length of each track or the number of musicians. We're failing at reusing common concepts and also leaving ourselves more code to test and maintain. We can shorten and tighten this code by rewriting it using the `Stream` abstraction and the Java 8 collections library. Example 7-6 is what we would come up with if we directly translated the imperative code to streams.

Example 7-6. A refactor of our imperative Order class to use streams

```
public long countRunningTime() {
    return albums.stream()
            .mapToLong(album -> album.getTracks()
                                     .mapToLong(track -> track.getLength())
                                     .sum())
```

```
            .sum();
}

public long countMusicians() {
    return albums.stream()
            .mapToLong(album -> album.getMusicians().count())
            .sum();
}

public long countTracks() {
    return albums.stream()
            .mapToLong(album -> album.getTracks().count())
            .sum();
}
```

It still suffers from the same reuse and readability issues, because there are certain abstractions and commonalities that are only expressible in domain terms. The streams library won't provide a method for you to count the number of a certain thing per album—that's the kind of domain method that you should be writing yourself. It's also the kind of domain method that was very hard to write before Java 8 because it's doing a different thing for each method.

Let's think about how we're going to implement such a function. We're going to return a `long` with the count of some feature for all the albums. We also need to take in some kind of lambda expression that tells us what the number for each album is. This means we need a method parameter that returns us a `long` for each album; conveniently, there is already a `ToLongFunction` in the Java 8 core libraries. As shown in Figure 7-1, it is parameterized by its argument type, so we're using `ToLongFunction<Album>`.

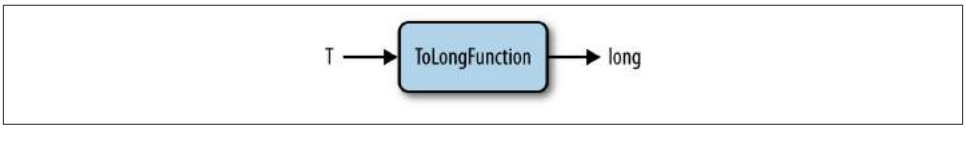

Figure 7-1. ToLongFunction

Now that we've made these decisions, the body of the method follows naturally. We take a `Stream` of the albums, map each album to a `long`, and then sum them. When we implement the consumer-facing methods such as `countTracks`, we pass in a lambda expression with behavior specific to that domain method. In this case, we're mapping the album to the number of tracks. Example 7-7 is what our code looks like when we've converted the code to use this domain-appropriate method.

Example 7-7. A refactor of our Order class to use domain-level methods

```
public long countFeature(ToLongFunction<Album> function) {
    return albums.stream()
            .mapToLong(function)
```

```
            .sum();
}

public long countTracks() {
    return countFeature(album -> album.getTracks().count());
}

public long countRunningTime() {
    return countFeature(album -> album.getTracks()
                                      .mapToLong(track -> track.getLength())
                                      .sum());
}

public long countMusicians() {
    return countFeature(album -> album.getMusicians().count());
}
```

Unit Testing Lambda Expressions

Unit testing is a method of testing individual chunks of code to ensure that they are behaving as intended.

Usually, when writing a unit test you call a method in your test code that gets called in your application. Given some inputs and possibly test doubles, you call these methods to test a certain behavior happening and then specify the changes you expect to result from this behavior.

Lambda expressions pose a slightly different challenge when unit testing code. Because they don't have a name, it's impossible to directly call them in your test code.

You could choose to copy the body of the lambda expression into your test and then test that copy, but this approach has the unfortunate side effect of not actually testing the behavior of your implementation. If you change the implementation code, your test will still pass even though the implementation is performing a different task.

There are two viable solutions to this problem. The first is to view the lambda expression as a block of code within its surrounding method. If you take this approach, you should be testing the behavior of the surrounding method, not the lambda expression itself. Let's take look Example 7-8, which gives an example method for converting a list of strings into their uppercase equivalents.

Example 7-8. Converting strings into their uppercase equivalents

```
public static List<String> allToUpperCase(List<String> words) {
    return words.stream()
```

```
                .map(string -> string.toUpperCase())
                .collect(Collectors.<String>toList());
}
```

The only thing that the lambda expression in this body of code does is directly call a core Java method. It's really not worth the effort of testing this lambda expression as an independent unit of code at all, since the behavior is so simple.

If I were to unit test this code, I would focus on the behavior of the method. For example, Example 7-9 is a test that if there are multiple words in the stream, they are all converted to their uppercase equivalents.

Example 7-9. Testing conversion of words to uppercase equivalents

```
@Test
public void multipleWordsToUppercase() {
    List<String> input = Arrays.asList("a", "b", "hello");
    List<String> result = Testing.allToUpperCase(input);
    assertEquals(asList("A", "B", "HELLO"), result);
}
```

Sometimes you want to use a lambda expression that exhibits complex functionality. Perhaps it has a number of corner cases or a role involving calculating a highly important function in your domain. You really want to test for behavior specific to that body of code, but it's in a lambda expression and you've got no way of referencing it.

As an example problem, let's look at a method that is slightly more complex than converting a list of strings to uppercase. Instead, we'll be converting the first character of a string to uppercase and leaving the rest as is. If we were to write this using streams and lambda expressions, we might write something like Example 7-10. Our lambda expression doing the conversion is at ❶.

Example 7-10. Convert first character of all list elements to uppercase

```
public static List<String> elementFirstToUpperCaseLambdas(List<String> words) {
    return words.stream()
            .map(value -> {  ❶
                char firstChar = Character.toUpperCase(value.charAt(0));
                return firstChar + value.substring(1);
            })
            .collect(Collectors.<String>toList());
}
```

Should we want to test this, we'd need to fire in a list and test the output for every single example we wanted to test. Example 7-11 provides an example of how cumbersome this approach becomes. Don't worry—there is a solution!

Example 7-11. Testing that in a two-character string, only the first character is converted to uppercase

```
@Test
public void twoLetterStringConvertedToUppercaseLambdas() {
    List<String> input = Arrays.asList("ab");
    List<String> result = Testing.elementFirstToUpperCaseLambdas(input);
    assertEquals(asList("Ab"), result);
}
```

Don't use a lambda expression. I know that might appear to be strange advice in a book about how to use lambda expressions, but square pegs don't fit into round holes very well. Having accepted this, we're bound to ask how we can still unit test our code and have the benefit of lambda-enabled libraries.

Do use method references. Any method that would have been written as a lambda expression can also be written as a normal method and then directly referenced elsewhere in code using method references.

In Example 7-12 I've refactored out the lambda expression into its own method. This is then used by the main method, which deals with converting the list of strings.

Example 7-12. Converting the first character to uppercase and applying it to a list

```
public static List<String> elementFirstToUppercase(List<String> words) {
    return words.stream()
                .map(Testing::firstToUppercase)
                .collect(Collectors.<String>toList());
}

public static String firstToUppercase(String value) {  ❶
    char firstChar = Character.toUpperCase(value.charAt(0));
    return firstChar + value.substring(1);
}
```

Having extracted the method that actually performs string processing, we can cover all the corner cases by testing that method on its own. The same test case in its new, simplified form is shown in Example 7-13.

Example 7-13. The two-character test applied to a single method

```
@Test
public void twoLetterStringConvertedToUppercase() {
    String input = "ab";
    String result = Testing.firstToUppercase(input);
    assertEquals("Ab", result);
}
```

Using Lambda Expressions in Test Doubles

A pretty common part of writing unit tests is to use *test doubles* to describe the expected behavior of other components of the system. This is useful because unit testing tries to test a class or method in isolation of the other components of your code base, and test doubles allow you to implement this isolation in terms of tests.

Even though test doubles are frequently referred to as *mocks*, actually both stubs and mocks are types of test double. The difference is that mocks allow you to verify the code's behavior. The best place to understand more about this is Martin Fowler's article on the subject.

One of the simplest ways to use lambda expressions in test code is to implement lightweight stubs. This is really easy and natural to implement if the collaborator to be stubbed is already a functional interface.

In "Behavioral Write Everything Twice" on page 99, I discussed how to refactor our common domain logic into a `countFeature` method that used a lambda expression to implement different counting behavior. Example 7-14 shows how we might go about unit testing part of its behavior.

Example 7-14. Using a lambda expression as a test double by passing it to countFeature

```
@Test
public void canCountFeatures() {
    OrderDomain order = new OrderDomain(asList(
            newAlbum("Exile on Main St."),
            newAlbum("Beggars Banquet"),
            newAlbum("Aftermath"),
            newAlbum("Let it Bleed")));

    assertEquals(8, order.countFeature(album -> 2));
}
```

The expected behavior is that the `countFeature` method returns the sum of some number for each album it's passed. So here I'm passing in four different albums, and the stub in my test is returning a count of 2 features for each album. I assert that the method returns 8—that is, 2×4. If you expect to pass a lambda expression into your code, then it's usually the right thing to have your test also pass in a lambda expression.

Most test doubles end up being the result of more complex expectation setting. In these situations, frameworks such as *Mockito* are often used to easily generate test doubles. Let's consider a simple example in which we want to produce a test double for a `List`. Instead of returning the size of the `List`, we want to return the size of another `List`. When mocking the `size` method of the `List`, we don't want to specify just a single

answer. We want our answer to perform some operation, so we pass in a lambda expression (Example 7-15).

Example 7-15. Using a lambda expression in conjunction with the Mockito library

```
List<String> list = mock(List.class);

when(list.size()).thenAnswer(inv -> otherList.size());

assertEquals(3, list.size());
```

Mockito uses an `Answer` interface that lets you provide alternative implementation behavior. In other words, it already supports our familiar friend: passing code as data. We can use a lambda expression here because `Answer` is, conveniently, a functional interface.

Lazy Evaluation Versus Debugging

Using a debugger typically involves stepping through statements of your program or attaching breakpoints. Sometimes you might encounter situations using the streams library where debugging becomes a little bit more complex, because the iteration is controlled by the library and many stream operations are lazily evaluated.

In the traditional imperative view of the world, in which code is a sequence of actions that achieve a goal, introspecting state after or before an action makes perfect sense. In Java 8, you still have access to all your existing IDE debugging tools, but sometimes you need to tweak your approach a little in order to achieve good results.

Logging and Printing

Let's say you're performing a series of operations on a collection and you're trying to debug the code; you want to see what the result of an individual operation is. One thing you could do is print out the collection value after each step. This is pretty hard with the Streams framework, as intermediate steps are lazily evaluated.

Let's take a look at how we might log intermediate values by taking a look at an imperative version of our nationality report from Chapter 3. In case you've forgotten, and who doesn't sometimes, this is code that tries to find the country of origin for every artist on an album. In Example 7-16 we're going to log each of the nationalities that we find.

Example 7-16. Logging intermediate values in order to debug a for loop

```
Set<String> nationalities = new HashSet<>();
for (Artist artist : album.getMusicianList()) {
    if (artist.getName().startsWith("The")) {
        String nationality = artist.getNationality();
        System.out.println("Found nationality: " + nationality);
        nationalities.add(nationality);
    }
```

```
}
return nationalities;
```

Now we could use the `forEach` method to print out the values from the stream, which would also cause it to be evaluated. However, this way has the downside that we can't continue to operate on that stream, because streams can only be used once. If we really want to use this approach, we need to recreate the stream. Example 7-17 shows how ugly this can get.

Example 7-17. Using a naive forEach to log intermediate values

```
album.getMusicians()
     .filter(artist -> artist.getName().startsWith("The"))
     .map(artist -> artist.getNationality())
     .forEach(nationality -> System.out.println("Found: " + nationality));

Set<String> nationalities
    = album.getMusicians()
           .filter(artist -> artist.getName().startsWith("The"))
           .map(artist -> artist.getNationality())
           .collect(Collectors.<String>toSet());
```

The Solution: peek

Fortunately, the streams library contains a method that lets you look at each value in turn and also lets you continue to operate on the same underlying stream. It's called `peek`. In Example 7-18, we have rewritten the previous example using `peek` in order to print out the stream values without having to repeat the pipeline of stream operations.

Example 7-18. Using peek to log intermediate values

```
Set<String> nationalities
    = album.getMusicians()
           .filter(artist -> artist.getName().startsWith("The"))
           .map(artist -> artist.getNationality())
           .peek(nation -> System.out.println("Found nationality: " + nation))
           .collect(Collectors.<String>toSet());
```

It's also possible to use the `peek` method to output to existing logging systems such as `log4j`, `java.util.logging`, or `slf4j` in exactly the same way.

Midstream Breakpoints

Logging is just one of many tricks that the `peek` method has up its sleeve. To allow us to debug a stream element by element, as we might debug a loop step by step, a breakpoint can be set on the body of the `peek` method.

In this case, peek can just have an empty body that you set a breakpoint in. Some debuggers won't let you set a breakpoint in an empty body, in which case I just map a value to itself in order to be able to set the breakpoint. It's not ideal, but it works fine.

Key Points

- Consider how lambda expressions can help when refactoring legacy code: there are common patterns.
- If you want to unit test a lambda expression of any complexity, extract it to a regular method.
- The peek method is very useful for logging out intermediate values when debugging.

CHAPTER 8
Design and Architectural Principles

The critical design tool for software development is a mind well educated in design principles. It is not…technology.

— Craig Larman

I've already established that lambda expressions are a fairly simple change to the Java language and that there are a bunch of ways that we can use them within the standard JDK libraries. Most Java code isn't written by the core JDK developers—it's written by people like you. In order to use lambda expressions in the most beneficial way possible, you need to start introducing them into your existing code base. They are just another tool in the belt of a professional Java developer, no different from an interface or a class.

In this chapter, we're going to explore how to use to use lambda expressions to implement the *SOLID* principles that provide guidelines toward good object-oriented programming. There are also many existing design patterns that can be improved by the use of lambda expressions, and we'll take a look at a smattering of those.

When coding with teammates at work, I'm sure you've come across a situation where you've implemented some feature or fixed a bug, and you were pretty happy with the way that you had done it, but soon after someone else took a look at the same code—perhaps during a code review—and they weren't so happy with it! It's pretty common to have this kind of disagreement over what really constitutes good code or bad code.

Most of the time when people are disagreeing, they are pushing a matter of opinion. The reviewer would have done it another way. It's not necessarily that he is right or wrong, or that you are right or wrong. When you welcome lambdas into your life, there's another topic to think about. It's not that they are a difficult feature or a big point of contention as much as they are just another design issue that people can discuss or disagree on.

This chapter is here to help! I'll try to put forth some well-grounded principles and patterns upon which you can compose maintainable and reliable software—not just to use the shiny new JDK libraries, but to use lambda expressions in your own domain architecture and applications.

Lambda-Enabled Design Patterns

One of the other bastions of design we're all familiar with is the idea of *design patterns*. Patterns document reusable templates that solve common problems in software architecture. If you spot a problem and you're familiar with an appropriate pattern, then you can take the pattern and apply it to your situation. In a sense, patterns codify what people consider to be a best-practice approach to a given problem.

Of course, no practice is ever the best practice forever. A common example is the once-popular singleton pattern, which enforces the creation of only one instance of an object. Over the last decade, this has been roundly criticized for making applications more brittle and harder to test. As the Agile software movement has made testing of applications more important, the issues with the singleton pattern have made it an *antipattern*: a pattern you should never use.

In this section, I'm not going to talk about how patterns have become obsolete. We're instead going to look at how existing design patterns have become better, simpler, or in some cases implementable in a different way. In all cases, the language changes in Java 8 are the driving factor behind the pattern changing.

Command Pattern

A *command object* is an object that encapsulates all the information required to call another method later. The *command pattern* is a way of using this object in order to write generic code that sequences and executes methods based on runtime decisions. There are four classes that take part in the command pattern, as shown in Figure 8-1:

Receiver
: Performs the actual work

Command
: Encapsulates all the information required to call the receiver

Invoker
: Controls the sequencing and execution of one or more commands

Client
: Creates concrete command instances

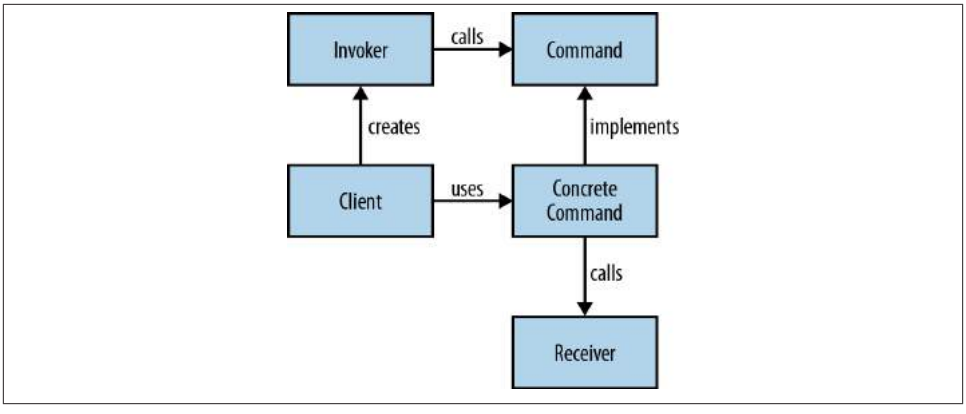

Figure 8-1. The command pattern

Let's look at a concrete example of the command pattern and see how it improves with lambda expressions. Suppose we have a GUI `Editor` component that has actions upon it that we'll be calling, such as `open` or `save`, like in Example 8-1. We want to implement macro functionality—that is, a series of operations that can be recorded and then run later as a single operation. This is our receiver.

Example 8-1. Common functions a text editor may have

```
public interface Editor {

    public void save();

    public void open();

    public void close();

}
```

In this example, each of the operations, such as `open` and `save`, are commands. We need a generic command interface to fit these different operations into. I'll call this interface `Action`, as it represents performing a single action within our domain. This is the interface that all our command objects implement (Example 8-2).

Example 8-2. All our actions implement the Action interface

```
public interface Action {

    public void perform();

}
```

We can now implement our `Action` interface for each of the operations. All these classes need to do is call a single method on our `Editor` and wrap this call into our `Action`

interface. I'll name the classes after the operations that they wrap, with the appropriate class naming convention—so, the `save` method corresponds to a class called `Save`. Examples 8-3 and 8-4 are our command objects.

Example 8-3. Our save action delegates to the underlying method call on Editor

```
public class Save implements Action {

    private final Editor editor;

    public Save(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void perform() {
        editor.save();
    }
}
```

Example 8-4. Our open action also delegates to the underlying method call on Editor

```
public class Open implements Action {

    private final Editor editor;

    public Open(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void perform() {
        editor.open();
    }
}
```

Now we can implement our `Macro` class. This class can `record` actions and run them as a group. We use a `List` to store the sequence of actions and then call `forEach` in order to execute each `Action` in turn. Example 8-5 is our invoker.

Example 8-5. A macro consists of a sequence of actions that can be invoked in turn

```
public class Macro {

    private final List<Action> actions;

    public Macro() {
        actions = new ArrayList<>();
    }

    public void record(Action action) {
        actions.add(action);
```

```
    }

    public void run() {
        actions.forEach(Action::perform);
    }

}
```

When we come to build up a macro programmatically, we add an instance of each command that has been recorded to the `Macro` object. We can then just run the macro and it will call each of the commands in turn. As a lazy programmer, I love the ability to define common workflows as macros. Did I say "lazy"? I meant focused on improving my productivity. The `Macro` object is our client code and is shown in Example 8-6.

Example 8-6. Building up a macro with the command pattern

```
Macro macro = new Macro();
macro.record(new Open(editor));
macro.record(new Save(editor));
macro.record(new Close(editor));
macro.run();
```

How do lambda expressions help? Actually, all our command classes, such as `Save` and `Open`, are really just lambda expressions wanting to get out of their shells. They are blocks of behavior that we're creating classes in order to pass around. This whole pattern becomes a lot simpler with lambda expressions because we can entirely dispense with these classes. Example 8-7 shows how to use our `Macro` class without these command classes and with lambda expressions instead.

Example 8-7. Using lambda expressions to build up a macro

```
Macro macro = new Macro();
macro.record(() -> editor.open());
macro.record(() -> editor.save());
macro.record(() -> editor.close());
macro.run();
```

In fact, we can do this even better by recognizing that each of these lambda expressions is performing a single method call. So, we can actually use method references in order to wire the editor's commands to the macro object (see Example 8-8).

Example 8-8. Using method references to build up a macro

```
Macro macro = new Macro();
macro.record(editor::open);
macro.record(editor::save);
macro.record(editor::close);
macro.run();
```

The command pattern is really just a poor man's lambda expression to begin with. By using actual lambda expressions or method references, we can clean up the code, reducing the amount of boilerplate required and making the intent of the code more obvious.

Macros are just one example of how we can use the command pattern. It's frequently used in implementing component-based GUI systems, undo functions, thread pools, transactions, and wizards.

There is already a functional interface with the same structure as our interface `Action` in core Java—`Runnable`. We could have chosen to use that in our macro class, but in this case it seemed more appropriate to consider an `Action` to be part of the vocabulary of our domain and create our own interface.

Strategy Pattern

The strategy pattern is a way of changing the algorithmic behavior of software based upon a runtime decision. How you implement the strategy pattern depends upon your circumstances, but in all cases the main idea is to be able to define a common problem that is solved by different algorithms and then encapsulate all the algorithms behind the same programming interface.

An example algorithm we might want to encapsulate is compressing files. We'll give our users the choice of compressing our files using either the *zip* algorithm or the *gzip* algorithm and implement a generic `Compressor` class that can compress using either algorithm.

First we need to define the API for our strategy (see Figure 8-2), which I'll call `CompressionStrategy`. Each of our compression algorithms will implement this interface. They have the `compress` method, which takes and returns an `OutputStream`. The returned `OutputStream` is a compressed version of the input (see Example 8-9).

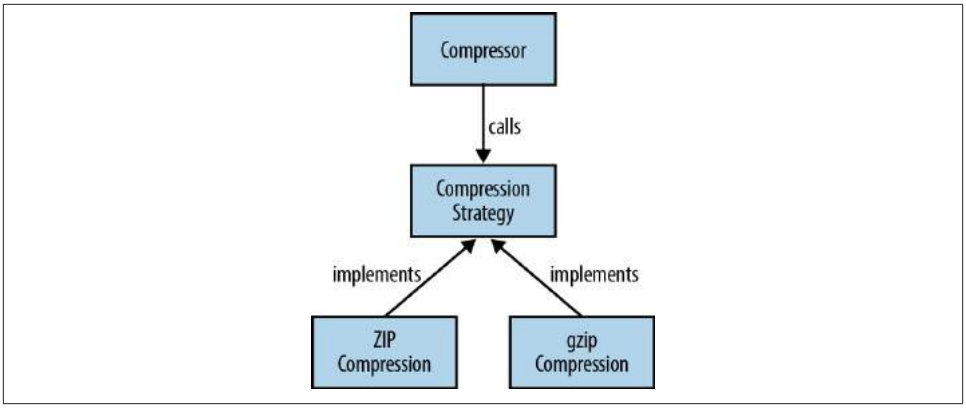

Figure 8-2. The strategy pattern

Example 8-9. Defining a strategy interface for compressing data

```
public interface CompressionStrategy {

    public OutputStream compress(OutputStream data) throws IOException;

}
```

We have two concrete implementations of this interface, one for gzip and one for ZIP, which use the built-in Java classes to write gzip (Example 8-10) and ZIP (Example 8-11) files.

Example 8-10. Using the gzip algorithm to compress data

```
public class GzipCompressionStrategy implements CompressionStrategy {

    @Override
    public OutputStream compress(OutputStream data) throws IOException {
        return new GZIPOutputStream(data);
    }

}
```

Example 8-11. Using the zip algorithm to compress data

```
public class ZipCompressionStrategy implements CompressionStrategy {

    @Override
    public OutputStream compress(OutputStream data) throws IOException {
        return new ZipOutputStream(data);
    }

}
```

Now we can implement our `Compressor` class, which is the *context* in which we use our strategy. This has a `compress` method on it that takes input and output files and writes a compressed version of the input file to the output file. It takes the `CompressionStrategy` as a constructor parameter that its calling code can use to make a runtime choice as to which compression strategy to use—for example, getting user input that would make the decision (see Example 8-12).

Example 8-12. Our compressor is provided with a compression strategy at construction time

```
public class Compressor {

    private final CompressionStrategy strategy;

    public Compressor(CompressionStrategy strategy) {
        this.strategy = strategy;
    }

    public void compress(Path inFile, File outFile) throws IOException {
        try (OutputStream outStream = new FileOutputStream(outFile)) {
            Files.copy(inFile, strategy.compress(outStream));
        }
    }
    }
```

If we have a traditional implementation of the strategy pattern, then we can write client code that creates a new `Compressor` with whichever strategy we want (Example 8-13).

Example 8-13. Instantiating the Compressor using concrete strategy classes

```
Compressor gzipCompressor = new Compressor(new GzipCompressionStrategy());
gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(new ZipCompressionStrategy());
zipCompressor.compress(inFile, outFile);
```

As with the command pattern discussed earlier, using either lambda expressions or method references allows us to remove a whole layer of boilerplate code from this pattern. In this case, we can remove each of the concrete strategy implementations and refer to a method that implements the algorithm. Here the algorithms are represented by the constructors of the relevant `OutputStream` implementation. We can totally dispense with the `GzipCompressionStrategy` and `ZipCompressionStrategy` classes when taking this approach. Example 8-14 is what the code would look like if we used method references.

Example 8-14. Instantiating the Compressor using method references

```
Compressor gzipCompressor = new Compressor(GZIPOutputStream::new);
gzipCompressor.compress(inFile, outFile);
```

```
Compressor zipCompressor = new Compressor(ZipOutputStream::new);
zipCompressor.compress(inFile, outFile);
```

Observer Pattern

The observer pattern is another behavioral pattern that can be improved and simplified through the use of lambda expressions. In the observer pattern, an object, called the *subject*, maintains a list of other objects, which are its *observers*. When the state of the subject changes, its observers are notified. It is heavily used in MVC-based GUI toolkits in order to allow view components to be updated when state changes in the model without coupling the two classes together.

Seeing GUI components update is a bit boring, so the subject that we'll be observing is the moon! Both NASA and some aliens want to keep track of things landing on the moon. NASA wants to make sure its Apollo astronauts have landed safely; the aliens want to invade Earth when NASA is distracted.

Let's start by defining the API of our observers, which I'll give the name `LandingObserver`. This has a single `observeLanding` method, which will be called when something lands on the moon (Example 8-15).

Example 8-15. An interface for observing organizations that land on the moon

```
public interface LandingObserver {

    public void observeLanding(String name);

}
```

Our subject class is the `Moon`, which keeps a list of `LandingObserver` instances, notifies them of landings, and can add new `LandingObserver` instances to spy on the `Moon` object (Example 8-16).

Example 8-16. Our Moon domain class—not as pretty as the real thing

```
public class Moon {

    private final List<LandingObserver> observers = new ArrayList<>();

    public void land(String name) {
        for (LandingObserver observer : observers) {
            observer.observeLanding(name);
        }
    }

    public void startSpying(LandingObserver observer) {
        observers.add(observer);
    }
    }
```

We have two concrete implementations of the `LandingObserver` class that represent the aliens' (Example 8-17) and NASA's views (Example 8-18) of the landing event. As mentioned earlier, they both have different interpretations of what this situation brings them.

Example 8-17. The aliens can observe people landing on the moon

```
public class Aliens implements LandingObserver {

    @Override
    public void observeLanding(String name) {
        if (name.contains("Apollo")) {
            System.out.println("They're distracted, lets invade earth!");
        }
    }

}
```

Example 8-18. NASA can also observe people landing on the moon

```
public class Nasa implements LandingObserver {
    @Override
    public void observeLanding(String name) {
        if (name.contains("Apollo")) {
            System.out.println("We made it!");
        }
    }
}
```

In a similar vein to the previous patterns, in our traditional example our client code specifically wires up a layer of boilerplate classes that don't need to exist if we use lambda expressions (see Examples 8-19 and 8-20).

Example 8-19. Client code building up a Moon using classes and things landing on it

```
Moon moon = new Moon();
moon.startSpying(new Nasa());
moon.startSpying(new Aliens());

moon.land("An asteroid");
moon.land("Apollo 11");
```

Example 8-20. Client code building up a Moon using lambdas and things landing on it

```
Moon moon = new Moon();

moon.startSpying(name -> {
    if (name.contains("Apollo"))
        System.out.println("We made it!");
});

moon.startSpying(name -> {
    if (name.contains("Apollo"))
```

```
        System.out.println("They're distracted, lets invade earth!");
});

moon.land("An asteroid");
moon.land("Apollo 11");
```

One thing to think about with both the observer and the strategy patterns is that whether to go down the lambda design route or the class route depends a lot on the complexity of the strategy or observer code that needs to be implemented. In the cases I've presented here, the code is simple in nature, just a method call or two, and fits the new language features well. In some situations, though, the observer can be a complex class in and of itself, and in those situations trying to squeeze a lot of code into one method can lead to poor readability.

In some respects, *trying to squeeze a lot of code into one method leads to poor readability* is the golden rule governing how to apply lambda expressions. The only reason I haven't pushed this so much is that it's also the golden rule of writing normal methods!

Template Method Pattern

A pretty common situation when developing software is having a common algorithm with a set of differing specifics. We want to require the different implementations to have a common pattern in order to ensure that they're following the same algorithm and also to make the code easier to understand. Once you understand the overall pattern, you can more easily understand each implementation.

The template method pattern is designed for these kinds of situations. Your overall algorithm design is represented by an *abstract class*. This has a series of abstract methods that represent customized steps in the algorithm, while any common code can be kept in this class. Each variant of the algorithm is implemented by a *concrete class* that overrides the abstract methods and provides the relevant implementation.

Let's think through a scenario in order to make this clearer. As a bank, we're going to be giving out loans to members of the public, companies, and employees. These categories have a fairly similar loan application process—you check the identity, credit history, and income history. You get this information from different sources and apply different criteria. For example, you might check the identity of a person by looking at an existing bill to her house, but companies have an official registrar such as the SEC in the US or Companies House in the UK.

We can start to model this in code with an abstract `LoanApplication` class that controls the algorithmic structure and holds common code for reporting the findings of the loan application. There are then concrete subclasses for each of our different categories of

applicant: `CompanyLoanApplication`, `PersonalLoanApplication`, and `EmployeeLoanApplication`. Example 8-21 shows what our `LoanApplication` class would look like.

Example 8-21. The process of applying for a loan using the template method pattern

```
public abstract class LoanApplication {

    public void checkLoanApplication() throws ApplicationDenied {
        checkIdentity();
        checkCreditHistory();
        checkIncomeHistory();
        reportFindings();
    }

    protected abstract void checkIdentity() throws ApplicationDenied;

    protected abstract void checkIncomeHistory() throws ApplicationDenied;

    protected abstract void checkCreditHistory() throws ApplicationDenied;

    private void reportFindings() {
```

`CompanyLoanApplication` implements `checkIdentity` by looking up information in a company registration database, such as Companies House. `checkIncomeHistory` would involve assessing existing profit and loss statements and balance sheets for the firm. `checkCreditHistory` would look into existing bad and outstanding debts.

`PersonalLoanApplication` implements `checkIdentity` by analyzing the paper statements that the client has been required to provide in order to check that the client's address exists. `checkIncomeHistory` involves assessing pay slips and checking whether the person is still employed. `checkCreditHistory` delegates to an external credit payment provider.

`EmployeeLoanApplication` is just `PersonalLoanApplication` with no employment history checking. Conveniently, our bank already checks all its employees' income histories when hiring them (Example 8-22).

Example 8-22. A special case of an an employee applying for a loan

```
public class EmployeeLoanApplication extends PersonalLoanApplication {

    @Override
    protected void checkIncomeHistory() {
        // They work for us!
    }

}
```

With lambda expressions and method references, we can think about the template method pattern in a different light and also implement it differently. What the template

method pattern is really trying to do is compose a sequence of method calls in a certain order. If we represent the functions as functional interfaces and then use lambda expressions or method references to implement those interfaces, we can gain a huge amount of flexibility over using inheritance to build up our algorithm. Let's look at how we would implement our `LoanApplication` algorithm this way, in Example 8-23!

Example 8-23. The special case of an employee applying for a loan

```
public class LoanApplication {

    private final Criteria identity;
    private final Criteria creditHistory;
    private final Criteria incomeHistory;

    public LoanApplication(Criteria identity,
                           Criteria creditHistory,
                           Criteria incomeHistory) {

        this.identity = identity;
        this.creditHistory = creditHistory;
        this.incomeHistory = incomeHistory;
    }

    public void checkLoanApplication() throws ApplicationDenied {
        identity.check();
        creditHistory.check();
        incomeHistory.check();
        reportFindings();
    }

    private void reportFindings() {
```

As you can see, instead of having a series of abstract methods we've got fields called `identity`, `creditHistory`, and `incomeHistory`. Each of these fields implements our `Criteria` functional interface. The `Criteria` interface checks a criterion and throws a domain exception if there's an error in passing the criterion. We could have chosen to return a domain class from the `check` method in order to denote failure or success, but continuing with an exception follows the broader pattern set out in the original implementation (see Example 8-24).

Example 8-24. A Criteria functional interface that throws an exception if our application fails

```
public interface Criteria {

    public void check() throws ApplicationDenied;

}
```

The advantage of choosing this approach over the inheritance-based pattern is that instead of tying the implementation of this algorithm into the `LoanApplication` hierarchy, we can be much more flexible about where to delegate the functionality to. For example, we may decide that our `Company` class should be responsible for all criteria checking. The `Company` class would then have a series of signatures like Example 8-25.

Example 8-25. The criteria methods on a Company

```
public void checkIdentity() throws ApplicationDenied;

public void checkProfitAndLoss() throws ApplicationDenied;

public void checkHistoricalDebt() throws ApplicationDenied;
```

Now all our `CompanyLoanApplication` class needs to do is pass in method references to those existing methods, as shown in Example 8-26.

Example 8-26. Our CompanyLoanApplication specifies which methods provide each criterion

```
public class CompanyLoanApplication extends LoanApplication {

    public CompanyLoanApplication(Company company) {
        super(company::checkIdentity,
              company::checkHistoricalDebt,
              company::checkProfitAndLoss);
    }

}
```

A motivating reason to delegate the behavior to our `Company` class is that looking up information about company identity differs between countries. In the UK, Companies House provides a canonical location for registering company information, but in the US this differs from state to state.

Using functional interfaces to implement the criteria doesn't preclude us from placing implementation within the subclasses, either. We can explicitly use a lambda expression to place implementation within these classes, or use a method reference to the current class.

We also don't need to enforce inheritance between `EmployeeLoanApplication` and `PersonalLoanApplication` to be able to reuse the functionality of `EmployeeLoanApplication` in `PersonalLoanApplication`. We can pass in references to the same methods. Whether they do genuinely subclass each other should really be determined by whether loans to employees are a special case of loans to people or a different type of loan. So, using this approach could allow us to model the underlying problem domain more closely.

Lambda-Enabled Domain-Specific Languages

A *domain-specific language* (DSL) is a programming language focused on a particular part of a software system. They are usually small and frequently less expressive than a general-purpose language, such as Java, for most programming tasks. DSLs are highly specialized: by trading off being good at everything, they get to be good at something.

It's usual to split up DSLs into two different categories: *internal* and *external*. An external DSL is one that is written separately from the source code of your program and then parsed and implemented separately. For example, Cascading Style Sheets (CSS) and regular expressions are commonly used external DSLs.

Internal DSLs are embedded into the programming language that they are written in. If you've used mocking libraries, such as JMock or Mockito, or a SQL builder API such as JOOQ or Querydsl, then you'll be familiar with internal DSLs. In one sense, they're just regular libraries that have an API designed to be fluent. Despite their simplicity, internal DSLs are valued because they can be a powerful tool for making your code more succinct and easier to read. Ideally, code written in a DSL reads like statements made within the problem domain that it is reflecting.

The introduction of lambda expressions makes it easier to implement DSLs that are fluent and adds another tool to the belt of those wanting to experiment in the DSL arena. We'll investigate those issues by building a DSL for performing behavior-driven development (BDD) called *LambdaBehave*.

BDD is a variant of test-driven development (TDD) that shifts the emphasis onto talking about the behavior of the program rather than simply the tests that it needs to pass. Our design is inspired by the JavaScript BDD framework Jasmine, which has been getting heavy use in frontend circles. Example 8-27 is a simple Jasmine suite that shows you how to create tests using Jasmine.

Example 8-27. Jasmine

```
describe("A suite is just a function", function() {
  it("and so is a spec", function() {
    var a = true;

    expect(a).toBe(true);
  });
});
```

I appreciate that if you're not familiar with JavaScript, this may seem confusing. We will be going through the concepts at a gentler pace as we build a Java 8 equivalent next. Just remember that the syntax for a lambda expression in JavaScript is `function() { … }`.

Let's take a look at each of the concepts in turn:

- Each *spec* describes a single behavior that your program exhibits.

- An *expectation* is a way of describing the behavior of the application. You will find expectations in specs.
- Groups of specs are combined into a *suite*.

Each of these concepts has an equivalent in a traditional testing framework, such as JUnit. A spec is similar to a test method, an expectation is similar to an assertion, and a suite is similar to a test class.

A DSL in Java

Let's look at an example of what we're aiming for with our Java-based BDD framework. Example 8-28 is a specification of some of the behaviors of a `Stack`.

Example 8-28. Some stories to specify a Stack

```
public class StackSpec {{

    describe("a stack", it -> {

        it.should("be empty when created", expect -> {
            expect.that(new Stack()).isEmpty();
        });

        it.should("push new elements onto the top of the stack", expect -> {
            Stack<Integer> stack = new Stack<>();
            stack.push(1);

            expect.that(stack.get(0)).isEqualTo(1);
        });

        it.should("pop the last element pushed onto the stack", expect -> {
            Stack<Integer> stack = new Stack<>();
            stack.push(2);
            stack.push(1);

            expect.that(stack.pop()).isEqualTo(2);
        });

    });

}}
```

We start off our suite of specifications using the `describe` verb. Then we give our suite a name that tells us what it's describing the behavior of; here, we've picked `"a stack"`.

Each of specifications reads as closely to an English sentence as possible. They all start with the prefix `it.should`, with `it` referring to the object whose behavior we're describing. There is then a plain English sentence that tells us what the behavior is that

we're thinking about. We can then describe expectations of the behavior of our object, which all start with the `expect.that` prefix.

When we check our specifications, we get a simple command-line report that tells us which pass or fail. You'll notice that "pop the last element pushed onto the stack" expected `pop` to be equal to `2`, not `1`, so it has failed:

```
a stack
    should pop the last element pushed onto the stack[expected:❶ but was:❷]
    should be empty when created
    should push new elements onto the top of the stack
```

How We Got There

So now that you've seen the kind of fluency we can get in our DSLs using lambda expressions, let's have a look at how I implemented the framework under the hood. Hopefully this will give you an idea of how easy it is to implement this kind of framework yourself.

The first thing I saw when I started describing behavior was the `describe` verb. This is really just a statically imported method. It creates an instance of our `Description` class for the suite and delegates handling the specification to it. The `Description` class corresponds to the `it` parameters in our specification language (see Example 8-29).

Example 8-29. The describe method that starts the definition of a specification

```
public static void describe(String name, Suite behavior) {
    Description description = new Description(name);
    behavior.specifySuite(description);
}
```

Each suite has its code description implemented by the user using a lambda expression. This means that we need a `Suite` functional interface, shown in Example 8-30, to represent a suite of specifications. You'll notice it also takes a `Description` object as an argument, which we passed into it from the `describe` method.

Example 8-30. Each suite of tests is a lambda expression implementing this interface

```
public interface Suite {

    public void specifySuite(Description description);

}
```

Not only are suites represented by lambda expressions in our DSL, but so are individual specifications. They also need a functional interface, which I'll call `Specification` (Example 8-31). The variable called `expect` in our code sample is an instance of our `Expect` class, which I'll describe later.

Example 8-31. Each specification is a lambda expression implementing this interface

```
public interface Specification {

    public void specifyBehaviour(Expect expect);

}
```

The `Description` instance we've been passing around comes in handy at this point. We want our users to be able to fluently name their specifications with the `it.should` clause. This means our `Description` class needs a `should` method (see Example 8-32). This is where the real work gets done, as this is the method that actually executes the lambda expression by calling its `specifySuite` method. Specifications will tell us they have failed by throwing the standard Java `AssertionError`, and we consider any other `Throwable` to be an error.

Example 8-32. Our specification lambda expressions get passed into the should method

```
public void should(String description, Specification specification) {
    try {
        Expect expect = new Expect();
        specification.specifyBehaviour(expect);
        Runner.current.recordSuccess(suite, description);
    } catch (AssertionError cause) {
        Runner.current.recordFailure(suite, description, cause);
    } catch (Throwable cause) {
        Runner.current.recordError(suite, description, cause);
    }
}
```

When our specifications want to describe an actual expectation, they use the `expect.that` clause. This means that our `Expect` class needs to have a method called `that` for users to call, shown in Example 8-33. This wraps up the object that gets passed in and can then expose fluent methods such as `isEqualTo` that throw the appropriate assertions if there's a specification failure.

Example 8-33. The start of the fluent expect chain

```
public final class Expect {

    public BoundExpectation that(Object value) {
        return new BoundExpectation(value);
    }

    // Rest of class omitted
```

You may have noticed one more detail that I've so far ignored and that has nothing to do with lambda expressions. Our `StackSpec` class didn't have any methods directly implemented on it, and I wrote the code inside. I've been a bit sneaky here and used double braces at the beginning and end of the class definition:

```
public class StackSpec {{
    ...
}}
```

These start an anonymous constructor that lets us execute an arbitrary block of Java code, so it's really just like writing out the constructor in full, but with a bit less boilerplate. I could have written the following instead:

```
public class StackSpec {
    public StackSpec() {
        ...
    }
}
```

There's a lot more work involved in implementing a complete BDD framework, but the purpose of this section is just to show you how to use lambda expressions to create more fluent domain-specific languages. I've covered the parts of the DSL that interact with lambda expressions in order to give you a flavor of how to implement this kind of DSL.

Evaluation

One aspect of fluency is the idea that your DSL is IDE-friendly. In other words, you can remember a minimal amount of knowledge and then use code completion to fill in the gaps in memory. This is why we use and pass it the `Description` and `Expect` objects. The other alternative would have been to have static imports for methods called `it` or `expect`, which is an approach used in some DSLs. If you pass the object into your lambda expression rather than requiring a static import, it makes it easier for a competent IDE user to code complete his way to working code.

The only thing a user needs to remember is the call to `describe`. The benefits of such an approach might not be obvious purely from reading this text, but I encourage you to test out the framework in a small sample project and see for yourself.

The other thing to notice is that most testing frameworks provide a bunch of annotations and use external *magic* or reflection. We didn't need to resort to such tricks. We can directly represent behavior in our DSLs using lambda expressions and treat these as regular Java methods.

Lambda-Enabled SOLID Principles

The SOLID principles are a set of basic principles for designing OO programs. The name itself is a acronym, with each of the five principles named after one of the letters: Single responsibility, Open/closed, Liskov substitution, Interface segregation, and Dependency inversion. The principles act as a set of guidelines to help you implement code that is easy to maintain and extend over time.

Each of the principles corresponds to a set of potential code smells that can exist in your code, and they offer a route out of the problems that they cause. Many books have been written on this topic, and I'm not going to cover the principles in comprehensive detail. I will, however, look at how three of the principles can be applied in the context of lambda expressions. In the Java 8 context, some of the principles can be extended beyond their original limitations.

The Single Responsibility Principle

Every class or method in your program should have only a single reason to change.

An inevitable fact of software development is that requirements change over time. Whether because a new feature needs to be added, your understanding of your problem domain or customer has changed, or you need things to be faster, over time software must evolve.

When the requirements of your software change, the responsibilities of the classes and methods that implement these requirements also change. If you have a class that has more than one responsibility, when a responsibility changes the resulting code changes can affect the other responsibilities that the class possesses. This possibly introduces bugs and also impedes the ability of the code base to evolve.

Let's consider a simple example program that generates a `BalanceSheet`. The program needs to tabulate the `BalanceSheet` from a list of assets and render the `BalanceSheet` to a PDF report. If the implementer chose to put both the responsibilities of tabulation and rendering into one class, then that class would have two reasons for change. You might wish to change the rendering in order to generate an alternative output, such as HTML. You might also wish to change the level of detail in the `BalanceSheet` itself. This is a good motivation to decompose this problem at the high level into two classes: one to tabulate the `BalanceSheet` and one to render it.

The single responsibility principle is stronger than that, though. A class should not just have a single responsibility: it should also encapsulate it. In other words, if I want to change the output format, then I should have to look at only the rendering class and not at the tabulation class.

This is part of the idea of a design exhibiting strong *cohesion*. A class is cohesive if its methods and fields should be treated together because they are closely related. If you tried to divide up a cohesive class, you would result in accidentally coupling the classes that you have just created.

Now that you're familiar with the single responsibility principle, the question arises, what does this have to do with lambda expressions? Well Lambda expressions make it a lot easier to implement the single responsibility principle at the method level. Let's take a look at some code that counts the number of prime numbers up to a certain value (Example 8-34).

Example 8-34. Counting prime numbers with multiple responsibilities in a method

```
public long countPrimes(int upTo) {
    long tally = 0;
    for (int i = 1; i < upTo; i++) {
        boolean isPrime = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                isPrime = false;
            }
        }
        if (isPrime) {
            tally++;
        }
    }
    return tally;
}
```

It's pretty obvious that we're really doing two things in Example 8-34:we're counting numbers with a certain property and we're checking whether a number is a prime. As shown in Example 8-35, we can easily refactor this to split apart these two responsibilities.

Example 8-35. Counting prime numbers after refactoring out the isPrime check

```
public long countPrimes(int upTo) {
    long tally = 0;
    for (int i = 1; i < upTo; i++) {
        if (isPrime(i)) {
            tally++;
        }
    }
    return tally;
}

private boolean isPrime(int number) {
    for (int i = 2; i < number; i++) {
        if (number % i == 0) {
            return false;
        }
    }
    return true;
}
```

Unfortunately, we're still left in a situation where our code has two responsibilities. For the most part, our code here is dealing with looping over numbers. If we follow the single responsibility principle, then iteration should be encapsulated elsewhere. There's also a good practical reason to improve this code. If we want to count the number of primes for a very large `upTo` value, then we want to be able to perform this operation in parallel. That's right—the threading model is a responsibility of the code!

We can refactor our code to use the Java 8 streams library (see Example 8-36), which delegates the responsibility for controlling the loop to the library itself. Here we use the `range` method to count the numbers between `0` and `upTo`, `filter` them to check that they really are prime, and then `count` the result.

Example 8-36. Refactoring the prime checking to use streams

```
public long countPrimes(int upTo) {
    return IntStream.range(1, upTo)
                    .filter(this::isPrime)
                    .count();
}

private boolean isPrime(int number) {
    return IntStream.range(2, number)
                    .allMatch(x -> (number % x) != 0);
}
```

If we want to speed up the time it takes to perform this operation at the expense of using more CPU resources, we can use the `parallelStream` method without changing any of the other code (see Example 8-37).

Example 8-37. The streams-based prime checking running in parallel

```
public long countPrimes(int upTo) {
    return IntStream.range(1, upTo)
                    .parallel()
                    .filter(this::isPrime)
                    .count();
}

private boolean isPrime(int number) {
    return IntStream.range(2, number)
             .allMatch(x -> (number % x) != 0);
}
```

So, we can use higher-order functions in order to help us easily implement the single responsibility principle.

The Open/Closed Principle

> *Software entities should be open for extension, but closed for modification.*
>
> — Bertrand Meyer

The overarching goal of the open/closed principle is similar to that of the single responsibility principle: to make your software less brittle to change. Again, the problem is that a single feature request or change to your software can ripple through the code base in a way that is likely to introduce new bugs. The open/closed principle is an effort

to avoid that problem by ensuring that existing classes can be extended without their internal implementation being modified.

When you first hear about the open/closed principle, it sounds like a bit of a pipe dream. How can you extend the functionality of a class without having to change its implementation? The actual answer is that you rely on an abstraction and can plug in new functionality that fits into this abstraction. Let's think through a concrete example.

We're writing a software program that measures information about system performance and graphs the results of these measurements. For example, we might have a graph that plots how much time the computer spends in user space, kernel space, and performing I/O. I'll call the class that has the responsibility for displaying these metrics `MetricDataGraph`.

One way of designing the `MetricDataGraph` class would be to have each of the new metric points pushed into it from the agent that gathers the data. So, its public API would look something like Example 8-38.

Example 8-38. The MetricDataGraph public API

```
class MetricDataGraph {

    public void updateUserTime(int value);

    public void updateSystemTime(int value);

    public void updateIoTime(int value);

}
```

But this would mean that every time we wanted to add in a new set of time points to the plot, we would have to modify the `MetricDataGraph` class. We can resolve this issue by introducing an abstraction, which I'll call a `TimeSeries`, that represents a series of points in time. Now our `MetricDataGraph` API can be simplified to not depend upon the different types of metric that it needs to display, as shown in Example 8-39.

Example 8-39. Simplified MetricDataGraph API

```
class MetricDataGraph {

    public void addTimeSeries(TimeSeries values);

}
```

Each set of metric data can then implement the `TimeSeries` interface and be plugged in. For example, we might have concrete classes called `UserTimeSeries`, `SystemTimeSeries`, and `IoTimeSeries`. If we wanted to add, say, the amount of CPU time that gets stolen from a machine if it's virtualized, then we would add a new implementation of

`TimeSeries` called `StealTimeSeries`. `MetricDataGraph` has been extended but hasn't been modified.

Higher-order functions also exhibit the same property of being open for extension, despite being closed for modification. A good example of this is the `ThreadLocal` class that we encountered earlier. The `ThreadLocal` class provides a variable that is special in the sense that each thread has a single copy for it to interact with. Its static `withInitial` method is a higher-order function that takes a lambda expression that represents a factory for producing an initial value.

This implements the open/closed principle because we can get new behavior out of `ThreadLocal` without modifying it. We pass in a different factory method to `withInitial` and get an instance of `ThreadLocal` with different behavior. For example, we can use `ThreadLocal` to produce a `DateFormatter` that is thread-safe with the code in Example 8-40.

Example 8-40. A ThreadLocal date formatter

```
// One implementation
ThreadLocal<DateFormat> localFormatter
    = ThreadLocal.withInitial(() -> new SimpleDateFormat());

// Usage
DateFormat formatter = localFormatter.get();
```

We can also generate completely different behavior by passing in a different lambda expression. For example, in Example 8-41 we're creating a unique identifier for each Java thread that is sequential.

Example 8-41. A ThreadLocal identifier

```
// Or...
AtomicInteger threadId = new AtomicInteger();
ThreadLocal<Integer> localId
    = ThreadLocal.withInitial(() -> threadId.getAndIncrement());

// Usage
int idForThisThread = localId.get();
```

Another interpretation of the open/closed principle that doesn't follow in the traditional vein is the idea that immutable objects implement the open/closed principle. An immutable object is one that can't be modified after it is created.

The term "immutability" can have two potential interpretations: *observable immutability* or *implementation immutability*. Observable immutability means that from the perspective of any other object, a class is immutable; implementation immutability means that the object never mutates. Implementation immutability implies observable immutability, but the inverse isn't necessarily true.

A good example of a class that proclaims its immutability but actually is only observably immutable is `java.lang.String`, as it caches the hash code that it computes the first time its `hashCode` method is called. This is entirely safe from the perspective of other classes because there's no way for them to observe the difference between it being computed in the constructor every time or cached.

I mention immutable objects in the context of a book on lambda expressions because they are a fairly familiar concept within functional programming, which is the same area that lambda expressions have come from. They naturally fit into the style of programming that I'm talking about in this book.

Immutable objects implement the open/closed principle in the sense that because their internal state can't be modified, it's safe to add new methods to them. The new methods can't alter the internal state of the object, so they are closed for modification, but they are adding behavior, so they are open to extension. Of course, you still need to be careful in order to avoid modifying state elsewhere in your program.

Immutable objects are also of particular interest because they are inherently thread-safe. There is no internal state to mutate, so they can be shared between different threads.

If we reflect on these different approaches, it's pretty clear that we've diverged quite a bit from the traditional open/closed principle. In fact, when Bertrand Meyer first introduced the principle, he defined it so that the class itself couldn't ever be altered after being completed. Within a modern Agile developer environment it's pretty clear that the idea of a class being complete is fairly outmoded. Business requirements and usage of the application may dictate that a class be used for something that it wasn't intended to be used for. That's not a reason to ignore the open/closed principle though, just a good example of how these principles should be taken as guidelines and heuristics rather than followed religiously or to the extreme.

A final point that I think is worth reflecting on is that in the context of Java 8, interpreting the open/closed principle as advocating an abstraction that we can plug multiple classes into or advocating higher-order functions amounts to the same thing. Because our abstraction needs to be represented by either an interface or an abstract class upon which methods are called, this approach to the open/closed principle is really just a usage of polymorphism.

In Java 8, any lambda expression that gets passed into a higher-order function is represented by a functional interface. The higher-order function calls its single method, which leads to different behavior depending upon which lambda expression gets passed in. Again, under the hood we're using polymorphism in order to implement the open/closed principle.

The Dependency Inversion Principle

Abstractions should not depend on details; details should depend on abstractions.

One of the ways in which we can make rigid and fragile programs that are resistant to change is by coupling high-level business logic and low-level code that is designed to glue modules together. This is because these are two different concerns that may change over time.

The goal of the dependency inversion principle is to allow programmers to write high-level business logic that is independent of low-level glue code. This allows us to reuse the high-level code in a way that is abstract of the details upon which it depends. This modularity and reuse goes both ways: we can substitute in different details in order to reuse the high-level code, and we can reuse the implementation details by layering alternative business logic on top.

Let's look at a concrete example of how the dependency inversion principle is traditionally used by thinking through the high-level decomposition involved in implementing an application that builds up an address book automatically. Our application takes in a sequence of electronic business cards as input and accumulates our address book in some storage mechanism.

It's fairly obvious that we can separate this code into three basic modules:

- The business card reader that understands an electronic business card format
- The address book storage that stores data into a text file
- The accumulation module that takes useful information from the business cards and puts it into the address book

We can visualize the relationship between these modules as shown in Figure 8-3.

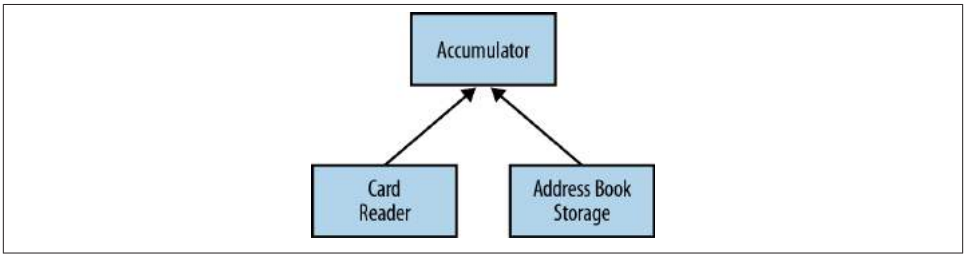

Figure 8-3. Dependencies

In this system, while reuse of the accumulation model is more complex, the business card reader and the address book storage do not depend on any other components. We

can therefore easily reuse them in another system. We can also change them; for example, we might want to use a different reader, such as reading from people's Twitter profiles, or we might want to store our address book in something other than a text file, such as a database.

In order to give ourselves the flexibility to change these components within our system, we need to ensure that the implementation of our accumulation module doesn't depend upon the specific details of either the business card reader or the address book storage. So, we introduce an abstraction for reading information and an abstraction for writing information. The implementation of our accumulation module depends upon these abstractions. We can pass in the specific details of these implementations at runtime. This is the dependency inversion principle at work.

In the context of lambda expressions, many of the higher-order functions that we've encountered enable a dependency inversion. A function such as `map` allows us to reuse code for the general concept of transforming a stream of values between different specific transformations. The `map` function doesn't depend upon the details of any of these specific transformations, but upon an abstraction. In this case, the abstraction is the functional interface `Function`.

A more complex example of dependency inversion is resource management. Obviously, there are lots of resources that can be managed, such as database connections, thread pools, files, and network connections. I'll use files as an example because they are a relatively simple resource, but the principle can easily be applied to more complex resources within your application.

Let's look at some code that extracts headings from a hypothetical markup language where each heading is designated by being suffixed with a colon (`:`). Our method is going to extract the headings from a file by reading the file, looking at each of the lines in turn, filtering out the headings, and then closing the file. We shall also wrap any `Exception` related to the file I/O into a friendly domain exception called a `HeadingLookupException`. The code looks like Example 8-42.

Example 8-42. Parsing the headings out of a file

```
public List<String> findHeadings(Reader input) {
    try (BufferedReader reader = new BufferedReader(input)) {
        return reader.lines()
                     .filter(line -> line.endsWith(":"))
                     .map(line -> line.substring(0, line.length() - 1))
                     .collect(toList());
    } catch (IOException e) {
        throw new HeadingLookupException(e);
    }
}
```

Unfortunately, our heading-finding code is coupled with the resource-management and file-handling code. What we really want to do is write some code that finds the headings and delegates the details of a file to another method. We can use a `Stream<String>` as the abstraction we want to depend upon rather than a file. A `Stream` is much safer and less open to abuse. We also want to be able to a pass in a function that creates our domain exception if there's a problem with the file. This approach, shown in Example 8-43, allows us to segregate the domain-level error handling from the resource management–level error handling.

Example 8-43. The domain logic with file handling split out

```
public List<String> findHeadings(Reader input) {
    return withLinesOf(input,
                       lines -> lines.filter(line -> line.endsWith(":"))
                                     .map(line -> line.substring(0, line.length()-1))
                                     .collect(toList()),
                       HeadingLookupException::new);
}
```

I expect that you're now wondering what that `withLinesOf` method looks like! It's shown in Example 8-44.

Example 8-44. The definition of withLinesOf

```
private <T> T withLinesOf(Reader input,
                          Function<Stream<String>, T> handler,
                          Function<IOException, RuntimeException> error) {

    try (BufferedReader reader = new BufferedReader(input)) {
        return handler.apply(reader.lines());
    } catch (IOException e) {
        throw error.apply(e);
    }
}
```

`withLinesOf` takes in a reader that handles the underlying file I/O. This is wrapped up in `BufferedReader`, which lets us read the file line by line. The `handler` function represents the body of whatever code we want to use with this function. It takes the `Stream` of the file's lines as its argument. We also take another handler called `error` that gets called when there's an exception in the I/O code. This constructs whatever domain exception we want. This exception then gets thrown in the event of a problem.

To summarize, higher-order functions provide an inversion of control, which is a form of dependency inversion. We can easily use them with lambda expressions. The other thing to note with the dependency inversion principle is that the abstraction that we depend upon doesn't have to be an `interface`. Here we've relied upon the existing `Stream` as an abstraction over raw reader and file handling. This approach also fits into the way that resource management is performed in functional languages—usually a

higher-order function manages the resource and takes a callback function that is applied to an open resource, which is closed afterward. In fact, if lambda expressions had been available at the time, it's arguable that the `try-with-resources` feature of Java 7 could have been implemented with a single library function.

Further Reading

A lot of the discussion in this chapter has delved into broader design issues, looking at the whole of your program rather than just local issues related to a single method. This is an area that we've just touched the surface of due to the lambda expressions focus of this book. There are a number of other books covering related topic areas that are worth investigating if you're interested in more detail.

The SOLID principles have long been emphasized by "Uncle" Bob Martin, who has both written and presented extensively on the topic. If you want to osmose some of his knowledge for free, a series of articles on each of the principles is available on the Object Mentor website, under the topic "Design Patterns."

If you are interested in a more comprehensive understanding of domain-specific languages, both internal and external, *Domain-Specific Languages* by Martin Fowler with Rebecca Parsons (Addison-Wesley) is recommended reading.

Key Points

- Lambda expressions can be used to make many existing design patterns simpler and more readable, especially the command pattern.
- There is more flexibility to the kind of domain-specific languages you can create with Java 8.
- New opportunities open up for applying the SOLID principles in Java 8.

CHAPTER 9
Lambda-Enabled Concurrency

I've already talked a bit about data parallelism, but in this chapter I'm going to cover how we can use lambda expressions to write concurrent applications that efficiently perform message passing and nonblocking I/O.

Some of the examples in this chapter are written using the Vert.x and RxJava frameworks. The principles are more general, though, and can be used in other frameworks and in your own code without you necessarily needing frameworks at all.

Why Use Nonblocking I/O?

When I introduced parallelism, I talked a lot about trying to use a lot of cores efficiently. That approach is really helpful, but when trying to process a lot of data, it's not the only threading model that you might want to use.

Let's suppose you're trying to write a chat service that handles a very high number of users. Every time a user connects to your service, a TCP connection to your server is opened. If you follow a traditional threading model, every time you want to write some data to your user, you would call a method that sends the user the data. This method call would block the thread that you're running on.

This approach to I/O, called *blocking I/O*, is fairly common and fairly easy to understand because the interaction with users follows a normal sequential flow of control through your program. The downside is that when you start looking at scaling your system to a significant number of users, you need to start a significant number of threads on your server in order to service them. That approach just doesn't scale well.

Nonblocking I/O—or, as it's sometimes called, *asynchronous I/O*—can be used to process many concurrent network connections without having an individual thread service each connection. Unlike with blocking I/O, the methods to read and write data to your chat clients return immediately. The actual I/O processing is happening in a separate thread,

and you are free to perform some useful work in the meantime. How you choose to use these saved CPU cycles may range from reading more data from another client to ticking over your Minecraft server!

I've so far avoided presenting any code to show the ideas because the concept of blocking versus nonblocking I/O can be implemented in a number of different ways in terms of the API. The Java standard library presents a nonblocking I/O API in the form of NIO (New I/O). The original version of NIO uses the concept of a `Selector`, which lets a thread manage multiple channels of communication, such as the network socket that's used to write to your chat client.

This approach never proved particularly popular with Java developers and resulted in code that was fairly hard to understand and debug. With the introduction of lambda expressions, it becomes idiomatic to design and develop APIs that don't have these deficiencies.

Callbacks

To demonstrate the principles of this approach, we're going to implement a dead simple chat application—no bells and no whistles. In this application, users can send and receive messages to and from each other. They are required to set names for themselves when they first connect.

We're going to implement the chat application using the Vert.x framework and introduce the necessary techniques as we go along. Let's start by writing some code that receives TCP connections, as demonstrated in Example 9-1.

Example 9-1. Receiving TCP connections

```
public class ChatVerticle extends Verticle {

    public void start() {
        vertx.createNetServer()
             .connectHandler(socket -> {
                 container.logger().info("socket connected");
                 socket.dataHandler(new User(socket, this));
             }).listen(10_000);

        container.logger().info("ChatVerticle started");
    }

}
```

You can think of a `Verticle` as being a bit like a `Servlet`—it's the atomic unit of deployment in the Vert.x framework. The entry point to the code is the `start` method, which is a bit like a `main` method in a regular Java program. In our chat app, we just use it to set up a server that accepts TCP connections.

We pass in a lambda expression to the `connectHandler` method, which gets called whenever someone connects to our chat app. This is a callback and works in a similar way to the Swing callbacks I talked about way back in Chapter 1. The benefit of this approach is that the application doesn't control the threading model—the Vert.x framework can deal with managing threads and all the associated complexity, and all we need to do is think in terms of events and callbacks.

Our application registers another callback using the `dataHandler` method. This is a callback that gets called whenever some data is read from the socket. In this case, we want to provide more complex functionality, so instead of passing in a lambda expression we use a regular class, `User`, and let it implement the necessary functional interface. The callback into our `User` class is listed in Example 9-2.

Example 9-2. Handling user connections

```
public class User implements Handler<Buffer> {

    private static final Pattern newline = Pattern.compile("\\n");

    private final NetSocket socket;
    private final Set<String> names;
    private final EventBus eventBus;

    private Optional<String> name;

    public User(NetSocket socket, Verticle verticle) {
        Vertx vertx = verticle.getVertx();

        this.socket = socket;
        names = vertx.sharedData().getSet("names");
        eventBus = vertx.eventBus();
        name = Optional.empty();
    }

    @Override
    public void handle(Buffer buffer) {
        newline.splitAsStream(buffer.toString())
               .forEach(line -> {
                   if (!name.isPresent())
                       setName(line);
                   else
                       handleMessage(line);
               });
    }

    // Class continues...
```

The `buffer` contains the data that has been written down the network connection to us. We're using a newline-separated, text-based protocol, so we want to convert this to a `String` and then split it based upon those newlines.

We have a regular expression to match newline characters, which is a `java.util.regex.Pattern` instance called `newline`. Conveniently, Java's `Pattern` class has had a `splitAsStream` method added in Java 8 that lets us split a `String` using the regular expression and have a stream of values, consisting of the values between each split.

The first thing our users do when they connect to our chat server is set their names. If we don't know the user's `name`, then we delegate to the logic for setting the name; otherwise, we handle our message like a normal chat message.

We also need a way of receiving messages from other users and passing them on to our chat client so the recipients can read them. In order to implement this, at the same time that we set the name of the current user we also register another callback that writes these messages (Example 9-3).

Example 9-3. Registering for chat messages

```
eventBus.registerHandler(name, (Message<String> msg) -> {
    sendClient(msg.body());
});
```

This code is actually taking advantage of Vert.x's *event bus*, which allows us to send messages between verticles in a nonblocking fashion (see Figure 9-1). The `registerHandler` method allows us to associate a handler with a certain address, so when a message is sent to that address the handler gets called with the message as its argument. Here we use the username as the address.

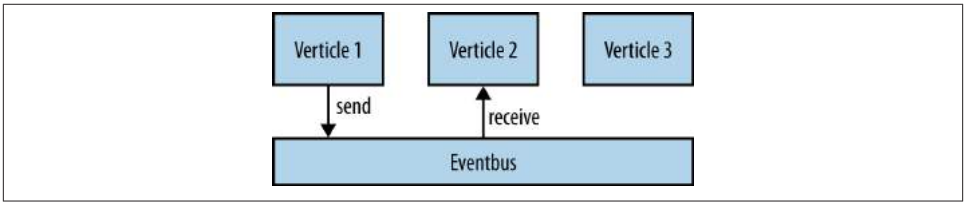

Figure 9-1. Event bus sending

By registering handlers at addresses and sending messages to them, it's possible to build up very sophisticated and/or decoupled sets of services that react in an entirely nonblocking fashion. Note that within our design, we share no state.

Vert.x's event bus lets us send a variety of types of message over it, but all of them will be wrapped in a `Message` object. Point-to-point messaging is available through the `Message` objects themselves; they may hold a reply handler for the sender of the `Message` object. Because in this case we want the actual body of the message—that is, the text itself—we just called the `body` method. We'll send this text message to the receiving user's chat client, implemented by writing the message down the TCP connection.

When our application wants to send a message from one user to another, it sends that message to the address that represents the other user (Example 9-4). Again, this is that user's username.

Example 9-4. Sending chat messages

```
eventBus.send(user, name.get() + '>' + message);
```

Let's extend this very basic chat server to broadcast messages and followers. There are two new commands that we need to implement in order for this to work:

- An exclamation mark representing the broadcast command, which sends all of its following text to any following users. For example, if bob typed "!hello followers", then all of his followers would receive "bob>hello followers".
- The follow command, which follows a specified user suffixed to the command, as in "follow bob".

Once we've parsed out the commands, we're going to implement the `broadcastMessage` and `followUser` methods, which correspond to each of these commands.

There's a different pattern of communication here as well. Instead of just having to send messages to a single user, you now have the ability to publish to multiple users. Fortunately, Vert.x's event bus also lets us publish a message to multiple handlers (see Figure 9-2). This lets us use a similar overarching approach.

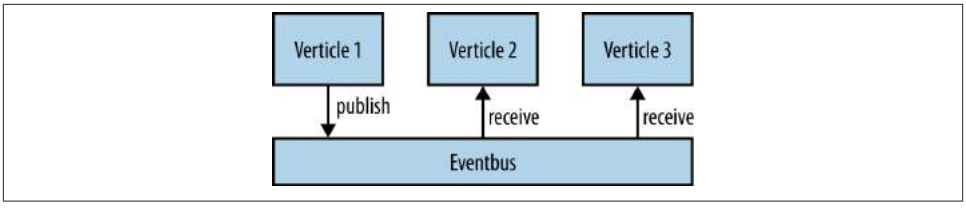

Figure 9-2. Event bus publishing

The only code difference is that we use the `publish` method on the event bus rather than the `send` method. To avoid overlapping with the existing addresses whenever a user uses the `!` command, it gets published to the user's name suffixed with `.followers`. So, for example, when bob publishes a message it goes to any handler registered on `bob.followers` (Example 9-5).

Example 9-5. Broadcasting messages to followers

```
private void broadcastMessage(String message) {
    String name = this.name.get();
    eventBus.publish(name + ".followers", name + '>' + message);
}
```

When it comes to the handler, we want to do the same operation that we performed earlier when registering sends: pass that message along to the client (Example 9-6).

Example 9-6. Receiving the broadcast messages

```
private void followUser(String user) {
    eventBus.registerHandler(user + ".followers", (Message<String> message) -> {
        sendClient(message.body());
    });
}
```

If you send to an address and there are multiple handlers listening on that address, a round-robin selector is used to decide which handler receives your message. This means that you need to be a bit careful when registering addresses.

Message Passing Architectures

What I've been describing here is a message passing–based architecture that I've implemented using a simple chat client. The details of the chat client are much less important than the overall pattern, so let's talk about message passing itself.

The first thing to note is that this is a no-shared-state design. All communication between our verticles is done by sending messages over our event bus. This means that we don't need to protect any shared state, so we don't need any kind of locks or use of the `synchronized` keyword in our code base at all. Concurrency is much simpler.

In order to ensure that we aren't sharing any state between verticles, we've actually imposed a few constraints on the types of messages being sent over the event bus. The example messages that we passed over the event bus in this case were plain old Java strings. These are immutable by design, which means that we can safely send them between verticles. Because the receiving handler can't modify the state of the `String`, it can't interfere with the behavior of the sender.

Vert.x doesn't restrict us to sending strings as messages, though; we can use more complex JSON objects or even build our own binary messages using the `Buffer` class. These aren't immutable messages, which means that if we just naively passed them around, our message senders and message handlers could share state by writing or reading through these messages.

The Vert.x framework avoids this problem by copying any mutable message the moment that you send it. That way the receiver gets the correct value, but you still aren't sharing state. Regardless of whether you're using the Vert.x framework, it's really important that you don't let your messages be an accidental source of shared state. Completely immutable messages are the simplest way of doing this, but copying the message also solves the problem.

The verticle model of development also lets us implement a concurrent system that is easy to test. This is because each verticle can be tested in isolation by sending messages in and expecting results to be returned. We can then compose a complex system out of individually tested components without incurring as many problems in integrating the components as we would if they were communicating via shared mutable state. Of course, end-to-end tests are still useful for making sure that your system does what your users expect of it!

Message passing–based systems also make it easier to isolate failure scenarios and write reliable code. If there is an error within a message handler, we have the choice of restarting its local verticle without having to restart the entire JVM.

In Chapter 6, we looked at how you can use lambda expressions in conjunction with the streams library in order to build data parallel code. That lets us use parallelism in order to process large amounts of data faster. Message passing and reactive programming, which we'll look at later in this chapter, are at the other end of the spectrum. We're looking at concurrency situations in which we want to have many more units of I/O work, such as connected chat clients, than we have threads running in parallel. In both cases, the solution is the same: use lambda expressions to represent the behavior and build APIs that manage the concurrency for you. Smarter libraries mean simpler application code.

The Pyramid of Doom

You've seen how we can use callbacks and events to produce nonblocking concurrent code, but I haven't mentioned the elephant in the room. If you write code with lots of callbacks, it becomes very hard to read, even with lambda expressions. Let's take a look at a more concrete example in order to understand this problem better.

While developing the chat server I wrote a series of tests that described the behavior of the verticle from the point of view of the client. The code for this is listed in the `messageFriend` test in Example 9-7.

Example 9-7. A test of whether two friends in our chat server can talk to each other

```
@Test
public void messageFriend() {
    withModule(() -> {
        withConnection(richard -> {
            richard.dataHandler(data -> {
                assertEquals("bob>oh its you!", data.toString());
                moduleTestComplete();
            });

            richard.write("richard\n");
            withConnection(bob -> {
                bob.dataHandler(data -> {
```

```
                    assertEquals("richard>hai", data.toString());
                    bob.write("richard<oh its you!");
                });
                bob.write("bob\n");
                vertx.setTimer(6, id -> richard.write("bob<hai"));
            });
        });
    });
}
```

I connect two clients, richard and bob, then richard says "hai" to bob and bob replies "oh it's you!" I've refactored out common code to make a connection, but even then you'll notice that the nested callbacks are beginning to turn into a *pyramid of doom*. They are stretching rightward across the screen, a bit like a pyramid sitting on its side (don't look at me—I didn't come up with the name!). This is a pretty well known anti-pattern, which makes it hard for a user to read and understand the code. It also stretches the logic of the code between multiple methods.

In the last chapter, I discussed how we could use lambda expressions to manage resources by passing a lambda expression into a `with` method. You'll notice in this test that I've used this pattern in couple of places. We've got a `withModule` method that deploys the current Vert.x module, runs some code, and shuts the module down. We've also got a `withConnection` method that connects to the `ChatVerticle` and then closes down the connection when it's done with it.

The benefit of using these `with` method calls here rather than using `try-with-resources` is that they fit into the nonblocking threading model that we're using in this chapter. We can try and refactor this code a bit in order to make it easier to understand, as in Example 9-8.

Example 9-8. A test of whether two friends in our chat server can talk to each other, split into different methods

```
@Test
public void canMessageFriend() {
    withModule(this::messageFriendWithModule);
}

private void messageFriendWithModule() {
    withConnection(richard -> {
        checkBobReplies(richard);
        richard.write("richard\n");
        messageBob(richard);
    });
}

private void messageBob(NetSocket richard) {
    withConnection(messageBobWithConnection(richard));
}
```

```
private Handler<NetSocket> messageBobWithConnection(NetSocket richard) {
    return bob -> {
        checkRichardMessagedYou(bob);
        bob.write("bob\n");
        vertx.setTimer(6, id -> richard.write("bob<hai"));
    };
}

private void checkRichardMessagedYou(NetSocket bob) {
    bob.dataHandler(data -> {
        assertEquals("richard>hai", data.toString());
        bob.write("richard<oh its you!");
    });
}

private void checkBobReplies(NetSocket richard) {
    richard.dataHandler(data -> {
        assertEquals("bob>oh its you!", data.toString());
        moduleTestComplete();
    });
}
```

The aggressive refactoring in Example 9-8 has solved the pyramid of doom problem, but at the expense of splitting up the logic of the single test into several methods. Instead of one method having a single responsibility, we have several sharing a responsibility between them! Our code is still hard to read, just for a different reason.

The more operations you want to chain or compose, the worse this problem gets. We need a better solution.

Futures

Another option when trying to build up complex sequences of concurrent operations is to use what's known as a `Future`. A `Future` is an IOU for a value. Instead of a method returning a value, it returns the `Future`. The `Future` doesn't have the value when it's first created, but it can be exchanged for the value later on, like an IOU.

You extract the value of the `Future` by calling its `get` method, which blocks until the value is ready. Unfortunately, `Futures` end up with composability issues, just like callbacks. We'll take a quick look at the issues that you can encounter.

The scenario we'll be considering is looking up information about an `Album` from some external web services. We need to find the list of tracks associated with a given `Album` and also a list of artists. We also need to ensure that we have sufficient credentials to access each of the services. So we need to log in, or at least make sure that we're already logged in.

Example 9-9 is an implementation of this problem using the existing `Future` API. We start out at ❶ by logging into the track and artist services. Each of these login actions returns a `Future<Credentials>` object with the login information. The `Future` interface is generic, so `Future<Credentials>` can be read as an IOU for a `Credentials` object.

Example 9-9. Downloading album information from some external web services using Futures

```
@Override
public Album lookupByName(String albumName) {
    Future<Credentials> trackLogin = loginTo("track");  ❶
    Future<Credentials> artistLogin = loginTo("artist");

    try {
        Future<List<Track>> tracks = lookupTracks(albumName, trackLogin.get());  ❷
        Future<List<Artist>> artists = lookupArtists(albumName, artistLogin.get());

        return new Album(albumName, tracks.get(), artists.get());  ❸
    } catch (InterruptedException | ExecutionException e) {
        throw new AlbumLookupException(e.getCause());  ❹
    }
}
```

At ❷ we make our calls to look up the tracks and artists given the login credentials and call `get` on both of these login credentials in order to get them out of the `Futures`. At ❸ we build up our new `Album` to return, again calling `get` in order to block on the existing `Futures`. If there's an exception, it gets thrown, so we have to propagate it through a domain exception at ❹.

As you'll have noticed, if you want to pass the result of one `Future` into the beginning of another piece of work, you end up blocking the thread of execution. This can be become a performance limitation because instead of work being executed in parallel it is (accidentally) run in serial.

What this means in the case of Example 9-9 is that we can't start either of the calls to the lookup services until we've logged into both of them. This is pretty unnecessary: `lookupTracks` only needs its login credentials, and `lookupArtists` should only need to wait for its login credentials. The breakdown of which actions need to wait for others to complete is shown in Figure 9-3.

We could take the blocking `get` calls and drag them down into the execution body of `lookupTracks` and `lookupArtists`. This would solve the problem, but would also result in uglier code and an inability to reuse credentials between multiple calls.

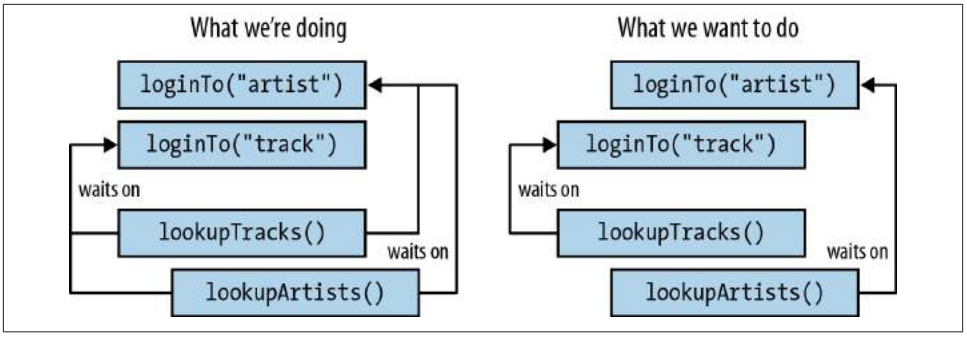

Figure 9-3. Both lookup actions don't need to wait for both login actions

What we really want here is a way of acting on the result of a `Future`, *without* having to make a blocking `get` call. We want to combine a `Future` with a callback.

Completable Futures

The solution to these issues arrives in the form of the `CompletableFuture`. This combines the IOU idea of a `Future` with using callbacks to handle event-driven work. The key point about the `CompletableFuture` is that you can compose different instances in a way that doesn't result in the pyramid of doom.

You might have encountered the concept behind the `CompletableFuture` before; various other languages call them a *deferred object* or a *promise*. In the Google Guava Library and the Spring Framework these are referred to as `ListenableFutures`.

I'll illustrate some usage scenarios by rewriting Example 9-9 to use `CompletableFuture`, rather than `Future`, as in Example 9-10.

Example 9-10. Downloading album information from some external web services using CompletableFutures

```
public Album lookupByName(String albumName) {
    CompletableFuture<List<Artist>> artistLookup
        = loginTo("artist")
         .thenCompose(artistLogin -> lookupArtists(albumName, artistLogin));    ❶

    return loginTo("track")
          .thenCompose(trackLogin -> lookupTracks(albumName, trackLogin))  ❷
          .thenCombine(artistLookup, (tracks, artists)
              -> new Album(albumName, tracks, artists))  ❸
          .join();  ❹
}
```

In Example 9-10 `loginTo`, `lookupArtists`, and `lookupTracks` all return a `CompletableFuture` instead of a `Future`. The key "trick" to the `CompletableFuture` API is to register lambda expressions and chain higher-order functions. The methods are different, but the concept is incredibly similar to the Streams API design.

At ❶ we use the `thenCompose` method to transform our `Credentials` into a `CompletableFuture` that contains the artists. This is a bit like taking an IOU for money from a friend and spending the money on Amazon when you get it. You don't immediately get a new book—you just get an email from Amazon saying that your book is on its way: a different form of IOU.

At ❷ we again use `thenCompose` and the `Credentials` from our Track API login in order to generate a `CompletableFuture` of tracks. We introduce a new method, `thenCombine`, at ❸. This takes the result from a `CompletableFuture` and combines it with another `CompletableFuture`. The combining operation is provided by the end user as a lambda expression. We want to take our tracks and artists and build up an `Album` object, so that's what we do.

At this point, it's worth reminding yourself that just like with the Streams API, we're not actually doing things; we're building up a recipe that says how to do things. Our method can't guarantee that the `CompletableFuture` has completed until one of the final methods is called. Because `CompletableFuture` implements `Future`, we could just call the `get` method again. `CompletableFuture` contains the `join` method, which is called at ❹ and does the same job. It doesn't have a load of the nasty checked exceptions that hindered our use of `get`.

You've probably got the basic idea of how to use `CompletableFuture`, but creating them is another matter. There are two different aspects of creating a `CompletableFuture`: creating the object itself and actually completing it by giving it the value that it owes its client code.

As Example 9-11 shows, it's pretty easy to create a `CompletableFuture` object. You just call its constructor! This object can now be handed out to client code for chaining operations. We also keep a reference to this object in order to process the work on another thread.

Example 9-11. Completing a future by providing a value

```
CompletableFuture<Artist> createFuture(String id) {
    CompletableFuture<Artist> future = new CompletableFuture<>();
    startJob(future);
    return future;
}
```

Once we've performed the work that needs to be done on whatever thread we're using, we need to tell the `CompletableFuture` what value it represents. Remember that this

work can by done through a number of different threading models. For example, we can `submit` a task to an `ExecutorService`, use an event loop-based system such as Vert.x, or just spin up a `Thread` and do work on it. As shown in Example 9-12, in order to tell the `CompletableFuture` that it's ready, you call the `complete` method. It's time to pay back the IOU.

Example 9-12. Completing a future by providing a value

```
future.complete(artist);
```

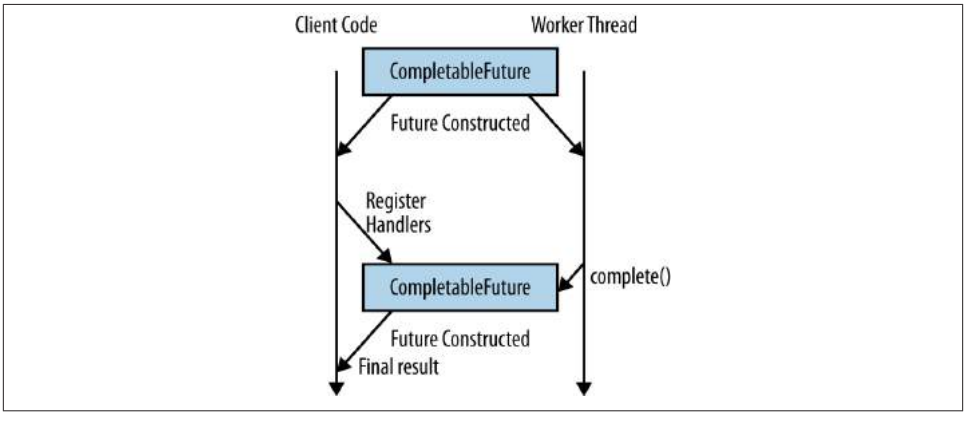

Figure 9-4. A completable future is an IOU that can be processed by handlers

Of course, a very common usage of `CompletableFuture` is to asynchronously run a block of code. This code completes and returns a value. In order to avoid lots of people having to implement the same code over and over again, there is a useful factory method for creating a `CompletableFuture`, shown in Example 9-13, called `supplyAsync`.

Example 9-13. Example code for asynchronously creating a CompletableFuture

```
CompletableFuture<Track> lookupTrack(String id) {
    return CompletableFuture.supplyAsync(() -> {
        // Some expensive work is done here ❶
        // ...
        return track;  ❷
    }, service);  ❸
}
```

The `supplyAsync` method takes a `Supplier` that gets executed. The key point, shown at ❶, is that this `Supplier` can do some time-consuming work without blocking the current thread—thus the *Async* in the method name. The value returned at ❷ is used to complete the `CompletableFuture`. At ❸ we provide an `Executor`, called `service`, that tells the

CompletableFuture where to perform the work. If no Executor is provided, it just uses the same fork/join thread pool that parallel streams execute on.

Of course, not every IOU has a happy ending. Sometimes we encounter exceptional circumstances and can't pay our debts. As Example 9-14 demonstrates, the CompletableFuture API accounts for these situations by letting you completeExceptionally. This can be called as an alternative to complete. You shouldn't call both complete and completeExceptionally on a CompletableFuture, though.

Example 9-14. Completing a future if there's an error

```
future.completeExceptionally(new AlbumLookupException("Unable to find " + name));
```

A complete investigation of the CompletableFuture API is rather beyond the scope of this chapter, but in many ways it is a hidden goodie bag. There are quite a few useful methods in the API for composing and combining different instances of CompletableFuture in pretty much any way imaginable. Besides, by now you should be familiar with the fluent style of chaining sequences of higher-order functions to tell the computer what to do.

Let's take a brief look at a few of those use cases:

- If you want to end your chain with a block of code that returns nothing, such as a Consumer or Runnable, then take a look at thenAccept and thenRun.
- Transforming the value of the CompletableFuture, a bit like using the map method on Stream, can be achieved using thenApply.
- If you want to convert situations in which your CompletableFuture has completed with an exception, the exceptionally method allows you to recover by registering a function to make an alternative value.
- If you need to do a map that takes account of both the exceptional case and regular use cases, use handle.
- When trying to figure out what is happening with your CompletableFuture, you can use the isDone and isCompletedExceptionally methods.

CompletableFuture is really useful for building up concurrent work, but it's not the only game in town. We're now going to look at a related concept that offers a bit more flexibility in exchange for more complex code.

Reactive Programming

The concept behind a CompletableFuture can be generalized from single values to complete streams of data using *reactive programming*. Reactive programming is actually

a form of declarative programming that lets us program in terms of changes and data flows that get automatically propagated.

You can think of a spreadsheet as a commonly used example of reactive programming. If you enter `=B1+5` in cell `C1`, it tells the spreadsheet to add `5` to the contents of cell `B1` and put the result in `C1`. In addition, the spreadsheet reacts to any future changes in `B1` and updates the value in `C1`.

The RxJava library is a port of these reactive ideas onto the JVM. We won't be going into the library in huge amounts of depth here, just covering the key concepts.

RxJava introduces a class called `Observable` that represents a sequence of events that you can react to. It's an IOU for a sequence. There is a strong connection between an `Observable` and the `Stream` interface that we encountered in Chapter 3.

In both cases we build up recipes for performing work by chaining higher-order functions and use lambda expressions in order to associate behavior with these general operations. In fact, many of the operations defined on an `Observable` are the same as on a `Stream`: `map`, `filter`, `reduce`.

The big difference between the approaches is the use case. Streams are designed to build up computational workflows over in-memory collections. RxJava, on the other hand, is designed to compose and sequence asynchronous and event-based systems. Instead of pulling data out, it gets pushed in. Another way of thinking about RxJava is that it is to a sequence of values what a `CompletableFuture` is to a single value.

Our concrete example this time around is searching for an artist and is shown in Example 9-15. The `search` method filters the results by name and nationality. It keeps a local cache of artist names but must look up other artist information, such as nationality, from external services.

Example 9-15. Searching for an artist by name and nationality

```
public Observable<Artist> search(String searchedName,
                                 String searchedNationality,
                                 int maxResults) {

    return getSavedArtists()   ❶
          .filter(name -> name.contains(searchedName))  ❷
          .flatMap(this::lookupArtist)  ❸
          .filter(artist -> artist.getNationality()  ❹
                                  .contains(searchedNationality))
          .take(maxResults);  ❺
}
```

At ❶ we get an `Observable` of the saved artist names. The higher-order functions on the `Observable` class are similar to those on the `Stream` interface, so at ❷ and ❹ we're

able to `filter` by artist name and nationality, in a similar manner as if we were using a `Stream`.

At ❸ we replace each name with its `Artist` object. If this were as simple as calling its constructor, we would obviously use the `map` operation. But in this case we need to compose a sequence of calls to external web services, each of which may be done in its own thread or on a thread pool. Consequently, we need to replace each name with an `Observable` representing one or more artists. So we use the `flatMap` operation.

We also need to limit ourselves to `maxResults` number of results in our search. To implement this at ❺, we call the `take` method on `Observable`.

As you can see, the API is quite stream-like in usage. The big difference is that while a `Stream` is designed to compute final results, the RxJava API acts more like `CompletableFuture` in its threading model.

In `CompletableFuture` we had to pay the IOU by calling `complete` with a value. Because an `Observable` represents a stream of events, we need the ability to push multiple values; Example 9-16 shows how to do this.

Example 9-16. Pushing values into an Observable and completing it

```
observer.onNext("a");
observer.onNext("b");
observer.onNext("c");
observer.onCompleted();
```

We call `onNext` repeatedly, once for each element in the `Observable`. We can do this in a loop and on whatever thread of execution we want to produce the values from. Once we have finished with whatever work is needed to generate the events, we call `onCompleted` to signal the end of the `Observable`. As well as the full-blown streaming approach, there are also several static convenience factory methods for creating `Observable` instances from futures, iterables, and arrays.

In a similar manner to `CompletableFuture`, the `Observable` API also allows for finishing with an exceptional event. We can use the `onError` method, shown in Example 9-17, in order to signal an error. The functionality here is a little different from `CompletableFuture`—you can still get all the events up until an exception occurs, but in both cases you either end normally or end exceptionally.

Example 9-17. Notifying your Observable that an error has occurred

```
observer.onError(new Exception());
```

As with `CompletableFuture`, I've only given a flavor of how and where to use the `Observable` API here. If you want more details, read the project's comprehensive documentation. RxJava is also beginning to be integrated into the existing ecosystem of Java

libraries. The enterprise integration framework Apache Camel, for example, has added a module called Camel RX that gives the ability to use RxJava with its framework. The Vert.x project has also started a project to Rx-ify its API.

When and Where

Throughout this chapter, I've talked about how to use nonblocking and event-based systems. Is that to say that everyone should just go out tomorrow and throw away their existing Java EE or Spring enterprise web applications? The answer is most definitely no.

Even accounting for `CompletableFuture` and RxJava being relatively new, there is still an additional level of complexity when using these idioms. They're simpler than using explicit futures and callbacks everywhere, but for many problems the traditional blocking web application development is just fine. If it ain't broke, don't fix it.

Of course, that's not to say that reading this chapter was a waste of your afternoon! Event-driven, reactive applications are growing in popularity and are frequently a great way to model the problems in your domain. The Reactive Manifesto advocates building more applications in this style, and if it's right for you, then you should. There are two scenarios in particular in which you might want to think in terms of reacting to events rather than blocking.

The first is when your business domain is phrased in terms of events. A classic example here is Twitter, a service for subscribing to streams of text messages. Your users are sending messages between one another, so by making your application event-driven, you are accurately modeling the business domain. Another example might be an application that tries to plot the price of shares. Each new price update can be modeled as an event.

The second obvious use case is a situation where your application needs to perform many I/O operations simultaneously. In these situations, performing blocking I/O requires too many threads to be spawned simultaneously. This results in too many locks in contention and too much context switching. If you want to deal with thousands of concurrent connections or more, it's usually better to go nonblocking.

Key Points

- Event-driven architectures are easy to implement using lambda-based callbacks.
- A `CompletableFuture` represents an IOU for a value. They can be easily composed and combined using lambdas.
- An `Observable` extends the idea of a `CompletableFuture` to streams of data.

Exercises

There's really only one exercise for this chapter, and it requires refactoring some code to use a `CompletableFuture`. We'll start out with the `BlockingArtistAnalyzer` class shown in Example 9-18. This class takes the names of two artists, looks up the `Artist` objects from the names, and returns `true` if the first artist has more members and `false` otherwise. It is injected with an `artistLookupService` that may take some time to look up the `Artist` in question. Because `BlockingArtistAnalyzer` blocks on this service twice sequentially, the analyzer can be slow; the goal of our exercise is to speed it up.

Example 9-18. The BlockingArtistAnalyzer tells its clients which Artist has more members

```
public class BlockingArtistAnalyzer {

    private final Function<String, Artist> artistLookupService;

    public BlockingArtistAnalyzer(Function<String, Artist> artistLookupService) {
        this.artistLookupService = artistLookupService;
    }

    public boolean isLargerGroup(String artistName, String otherArtistName) {
        return getNumberOfMembers(artistName) > getNumberOfMembers(otherArtistName);
    }

    private long getNumberOfMembers(String artistName) {
        return artistLookupService.apply(artistName)
                                  .getMembers()
                                  .count();
    }

}
```

The first part of this exercise is to refactor the blocking return code to use a callback interface. In this case, we'll be using a `Consumer<Boolean>`. Remember that `Consumer` is a functional interface that ships with the JVM that accepts a value and returns `void`. Your mission, should you choose to accept it, is to alter `BlockingArtistAnalyzer` so that it implements `ArtistAnalyzer` (Example 9-19).

Example 9-19. The ArtistAnalyzer that you need to make BlockingArtistAnalyzer implement

```
public interface ArtistAnalyzer {

    public void isLargerGroup(String artistName,
                              String otherArtistName,
                              Consumer<Boolean> handler);

}
```

Now that we have an API that fits into the callback model, we can remove the need for both of the blocking lookups to happen at the same time. You should refactor the `isLargerGroup` method so that they can operate concurrently using the `CompletableFuture` class.

CHAPTER 10
Moving Forward

In many ways, Java as a language has stood the test of time well. It's still an incredibly popular platform and a good choice for developing enterprise business software. A vast array of open source libraries and frameworks have been developed, solving every problem from how to write a modular and complex web application (Spring) right down to getting basic date and time arithmetic right (Jodatime). The tooling in space from IDEs such as Eclipse and Intellij right through to build systems like gradle and maven is unrivaled.

Unfortunately, over the years Java has acquired a bit of a reputation as a staid development choice that has failed to evolve with the times, in part because it has been popular for a long period of time; familiarity breeds contempt. And of course, there have been genuine issues around the evolution of Java. The decision to maintain backward compatibility, despite its benefits, has complicated this.

Fortunately, the arrival of Java 8 signals not just an incremental improvement in the language but a step change in its development. Unlike Java 6 and 7, this release doesn't equate to a few minor library improvements. I fully expect and hope that future releases of Java will continue the rapid pace of improvement seen with Java 8. That's not just because I've enjoyed writing a book on the topic! I really do think that there is a long way to go in terms of improving the fundamental task of programming: making code easier to read, clarifying its intent, making it easier to write high-performance code. My only regret is that there isn't enough space in this concluding chapter to detail the full potential of future releases.

We're nearing the end of this book, but I hope we're not nearing the end of your time with Java 8. I've covered a bunch of different ways you can use lambda expressions: better collections library code, data parallelism, simpler and cleaner code, and finally concurrency. I've explained the why, what, and how, but it is still up to you to put everything into practice. In this spirit, here is a series of open exercises to which there

are no right and wrong answers. Undertaking them can help guide your ongoing learning experience:

- Explain what lambda expressions are and why they should be of interest to another programmer. This could be a friend or a coworker.
- Start a trial deployment of your work product on Java 8. If you've already got your unit tests running under the Jenkins CI system, then it's very easy to run the same build under multiple Java versions.
- Start refactoring a bit of legacy code in a real product to use streams and collectors. This could be an open source project you're interested in or maybe even your work product if the trial deployment went well. If you aren't ready to move wholesale, then perhaps prototyping things on a different branch is a good way to start.
- Do you have any concurrency problems or large-scale data-processing code? If so, try to prototype a refactor in order to use either streams for data parallelism or some of the new concurrency features in RxJava or `CompletableFuture`.
- Have a look at the design and architecture of a code base you know really well:
  - — Could it be implemented better at a macro level?
  - — Can you simplify the design?
  - — Can you reduce the amount of code needed to implement a feature?
  - — Can the code be made easier to read?

Index

A

abstract classes, 119
accumulators, 28
ActionListener class, 5
Agile software development, 110
    immutable objects and, 133
Amdahls Law, 83
anonymous inner class, 5, 42
Answer interface (Mockito), 106
antipatterns, 110
Apache Camel, 155
Apache Maven, 159
arguments, 7
array operations, 92
    parallel performance of, 91
ArrayList data source, 91
Arrays class, 92
    parallelPrefix operation, 93
    parallelSetAll operation, 92
    parallelSort operation, 92
asynchronous I/O (see nonblocking I/O)
averagingInt as downstream collector, 69

B

backwards binary compatibility, 47
Beck, Kent, 97
behavior-driven development (BDD), 123
    test-driven development vs., 123
binary objects, passing between verticals, 144
BinaryOperator interface, 13, 45
blocking I/O, 139
boxed type, 42
boxed() method, 44
boxing, 43
Buffer class (vertx), 144
build method, 21
builder pattern, 21
bun methods, 22

C

callbacks, 140–144
    combining with Futures, 149–152
    Pyramid of Doom and, 145–147
Camel (Apache), 155
cascading style sheets, as external DSL, 123
class cohesion, 128
client class (command pattern), 110
Closeable interface, 47
closures, 9
code
    as data, 5
    opinions on, 109
cohesion, 128
collect(toList()) operation (Stream), 22
collections, 59–78
    collectors, 62–76
    creating parallel streams from, 83

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

in Java 8, 77–78
method references, 59
stream elements, ordering, 60–62
Collections interface, 48
collectors, 62–76
composing, 67–69
custom, 69–76
data partitioning with, 64
grouping data with, 65
into other collections, 62
reduction as, 76
refactoring, 69–76
strings, 66
to values, 63
combining functions, 88
command class (command pattern), 110
command pattern, 110–114
lambda expressions vs., 113
Comparable interface, 47
Comparator, 26
comparing method, 26
completable futures, 149–152
CompletableFuture API, 160
complete method, 151
completeExceptionally, 152
exceptionally method, 152
Executor, 151
isCompletedExceptionally method, 152
isDone method, 152
join method, 150
Streams API vs., 150
supplyAsync method, 151
thenAccept method, 152
thenApply method, 152
thenCombine method, 150
thenRun method, 152
complete method (CompletableFuture), 151
completeExceptionally (CompletableFuture), 152
concrete classes, 119
concurrency, 139–155
appropriate use of, 155
callbacks and, 140–144
completable futures, 149–152
futures, 147–152
message passing architecture, 144
nonblocking I/O, 139
parallelism vs., 81–83, 81
reactive programming, 152–155
concurrent package, 1
connectHandler method (vertx), 141
Consumer, 152
continuous integration (CI), 97
CSS, as external DSL, 123

D

data parallelism, 81–94
array operations for, 92
concurrency vs., 81–83, 81
performance and, 83
performance for, 89–92
rules of, 88
simulations and, 85–88
stream operations and, 83–85
task parallelism vs., 82
data size and performance, 89
dataHandler method (vertx), 141
debug(String message), 41
debugging vs. lazy evaluation, 106
default keyword, 48–52
subclassing and, 49–52
default methods, 48
deferred objects, 149
dependency inversion principle, 134–137
describe verb (DSL), 124
design patterns, 110
design principles, 109–137
command pattern, 110–114
domain-specific languages and, 123–127
lambda enabled, 110–122
observer pattern, 117–119
SOLID principles, 127–137
strategy pattern, 114–116
template method pattern, 119–122
diamond operator, 11
domain model, exposing/hiding elements of, 31
domain-specific languages (DSL), 123–127
describe verb, 124
evaluating, 127
expect class, 126
expectation, 124
external, 123
implementing, 125–127
in Java, 124
spec, 123
specifySuite method, 126
suite, 124
testing, 127

Domain-Specific Languages (Fowler and Parsons), 137
double type, 43

E

eager methods, 20
Eclipse, 159
effectively final variables, 8
empty method (Optional), 56
encounter order, 60
eventbus (vertx), 142
  broadcast messages, implementing with, 143
exceptionally method (CompletableFutures), 152
Executor (CompletableFuture), 151
ExecutorService, 151
expect class (DSL), 126
expectation (DSL), 124
external DSL, 123
external iteration, 17–20

F

filter method (Observable), 154
filter method (Stream), 24
final variables, 8
flatMap method (Observable), 154
flatmap operation, 25
for loops
  external iteration vs., 18
  refactoring, 32–34
forEach method (Iterable), 48
Fowler, Martin, 105, 137
Freeman, Steve, 97
Function objects
  flatmap operations and, 26
  passing lambda expressions as, 23
functional interfaces, 9–11
  for DSLs, 125
functional programming, 2
  higher-order functions and, 36
@FunctionalInterface annotation, 47
futures, 147–152
  completable, 149–152

G

get method (Futures), 42, 147
  Optional values and, 27, 56
gradle, 159
group by (SQL), 66
groupingBy collector, 65, 67
Growing Object-Oriented Software, Guided by Tests (Freeman and Pryce), 97
Guava Library (Google), 149
GUI Editor example, 111
gzip algorithm, 114

H

HashSet (Collection), 60
  in parallel processing, 91
higher-order functions, 36

I

identity values, 88
immutable objects, 132
implementation immutability, 132
inheritance, 52
  of default methods, 49–52
  rules of, 53
  super syntax and, 53
int type, 43
Intellij, 159
interfaces
  limitations of, 54
  static methods on, 54
  stream as, 54
internal DSL, 123
internal iteration, 17–20
IntStream.range constructor, 91
invoker class (command pattern), 110
isCompletedExceptionally method (CompletableFutures), 152
isDebugEnabled method, 98
  logging and, 41
isDone method (CompletableFutures), 152
isPresent method (Optional), 56
Iterable objects, 48
iterate method (Stream), 91
iteration
  external vs. internal, 17–20
  implementing, 20
iterator method (Streams), 18

J

Jasmine framework, 123

Java 6, 159
Java 7, 159
  type inference in, 12
Java 8, 1
  backwards binary compatibility, 47
  collections in, 77–78
  comparing method, 26
  default keyword, 48–52
  design principles, 109–137
  domain, 3
  functional programming in, 2
  @FunctionalInterface annotation, 47
  general patterns in, 27
  method references, 59
  multiple inheritance, 52
  NIO API, 140
  open/closed principle and, 133
  optional core library, 55
Jenkins CI system, 160
JMock, 123
Jodatime, 159
join method (CompletableFuture), 150
joining method (Collectors), 67
JOOQ, 123
JSON objects and verticals, 144
JUnit, 124

L

lambda expressions, 5–13
  best practices for, 36
  collections, 59–78
  command pattern vs., 113
  concurrency, 139–155
  design principles, 109–137
  domain-specific languages and, 123–127
  format of, 6–8
  functional interfaces and, 9–11
  iteration, 17–20
  libraries, 41–56
  observer pattern vs., 118
  reading, 7
  refactoring, 97–102
  strategy pattern vs., 116
  streams and, 17–37
  template method pattern vs., 120
  test doubles, using, 105–106
  type inference, 11–13
  unit testing, 102–104
  usage, 41
  values and, 8
lambdifiers, 97
lazy evaluation vs. debugging, 106
lazy methods, 20
libraries, 41–56
  @FunctionalInterface annotation, 47
  optional core, 55
  overloading methods, 45–47
  primitive types and, 42–45
lines method (BufferedReader), 91
LinkedList, 91
List (Collection), 60
ListenableFuture, 149
locking data structures, 88
log4j logging system, 41, 107
logging, 106
  speeding up with lambda expressions, 42
logging (Util), 107
long type, 43
LongStream function, 44
LongUnaryOperator function, 44

M

macro functionality, 111
macros, 112
Map collectors, 77
map method (Observable), 154
map operation (Stream), 22
  for specialized primitives, 44
mapping collector, 68
mapToLong function, 43
mapToObj method, 44
Martin, Bob, 137
Maven (Apache), 159
max operation (Stream), 26
maxBy collector, 63
Message object (vertx), 142
message passing architecture, 144
method references, 59
  unit testing and, 104
Meyer, Bertrand, 133
midstream breakpoints, 107
min operation (Stream), 26
minBy collector, 63
mocking libraries, 123
Mockito framework, 105, 123
mocks, 105
Mocks Arent Stubs (Fowler), 105
Monte Carlo simulations, 85–88

multicore CPUs, 1, 83
multiple inheritance, 52
  rules of, 53

N

New I/O (see nonblocking I/O)
newline (Pattern class), 142
NIO (see nonblocking I/O)
nonblocking I/O, 139
  blocking I/O vs., 139
  Java API for, 140
null value, 55
NullPointerException, 55

O

Object Mentor website, 137
Observable (RxJava), 153
  filter method, 154
  flatMap method, 154
  map method, 154
  onCompleted method, 154
  onError method, 154
  onNext method, 154
  Stream vs., 153
  take method, 154
observable immutability, 132
observer pattern, 117–119
  designing APIs for, 117
  lambda expressions vs., 118
ofNullable method (Optional), 56
onCompleted method (Observable), 154
onError method (Observable), 154
onNext method (Observable), 154
open/closed principle, 130–133
  immutable objects, 132
operations
  chaining, 30
  parallel array, 92
  parallel stream, 83–85
  refactoring legacy code, 31–34
optional core library, 55
Optional values, 27
  empty method, 56
  isPresent method, 56
  ofNullable method, 56
Oracle, 68
orElse method, 56
orElseGet method, 56
overloading methods, 45–47
  default methods and, 49–52
  ThreadLocal and, 98

P

parallel method (Stream), 83
parallelPrefix operation (Arrays), 93
parallelSetAll operation (Arrays), 92
parallelSort operation (Arrays), 92
parallelStream method (Collection), 83
parameter types, 9
Parsons, Rebecca, 137
partitioningBy collector, 64
Pattern class (Regex), 142
peek operation, 107
performance
  boxed types and, 43
  Futures and, 148
  logging and, 41
  parallelism and, 83, 89–92
  primitive specialized functions and, 44
  sequential vs. parallel processing and, 84
point lambdafication, 97
Predicate, 12
  filter method with, 25
  overloaded methods and, 46
  stream partitioning with, 64
primitive specialization, 43
primitive types, 42–45
  Stream specialization for, 43
promise, 149
Pryce, Nat, 97
Pyramid of Doom, 145–147

Q

Querydsl, 123

R

Reactive Manifesto website, 155
reactive programming, 152–155
readability of code, 2
  final values and, 8
  method references and, 60
receiver class (command pattern), 110
reduce operation (Stream), 28–30
  in parallel, 88
reduction, collectors as, 76

refactoring code, 97–102
  logging, 98
  overriding single methods, 98
  WET pattern and, 99–102
  with collectors, 69–76
refactoring legacy code, 31–34
regular expressions, as external DSL, 123
Runnable, 114
  CompletableFutures and, 152
RxJava framework, 139, 160
  documentation, 154
  reactive programming with, 153

S

Selectors, 140
sequential method (Stream), 88
Servlets vs. Verticals, 140
side effect-free functions, 36
single responsibility principle, 128–130
singleton pattern, 110
slf4j logging system, 41, 107
SOLID principles, 127–137
  dependency inversion principle, 134–137
  open/closed principle, 130–133
  single responsibility principle, 128–130
spec (DSL), 123
specifySuite method (DSL), 126
splitAsStream (Pattern class), 142
Spring, 159
Spring Framework, 149
SQL builder APIs, 123
state
  sharing between verticals, avoiding, 144
  streams and, 91
stateful stream operations, 91
stateless stream operations, 91
static methods, 54
strategy pattern, 114–116
  lambda expressions vs., 116
Stream API, 21–31
  collect(toList()) operation, 22
  CompletableFuture API vs., 150
  filter method, 24
  iterate method, 91
  map operation, 22, 44
  max operation, 26
  min operation, 26
  parallel method, 83
  peek, 107
  reduce operation, 28–30, 88
  RxJava vs., 153
  sequential method, 88
  unordered() method, 62
stream method, backwards compatibility of, 48
streams, 17–37
  chaining operations on, 30
  filter method, 24
  flatmap operation, 25
  implementing, 20
  iterating over, 17–20
  legacy code and, 31–34
  multiple calls to, 34
  operations, 21–31
  ordering elements in, 60–62
  specialized versions of, 44
  stateful operations, 91
  stateless operations, 91
String objects
  as immutable code, 133
  splitting with splitAsStream, 142
strings, 66
stubs, 105
subclassing, 49–52
subject (observer pattern), 117
suite (DSL), 124
summarizingInt collector, 64
summarizingLong, as downstream collector, 69
summaryStatistics method, 44
  summarizingInt collector, 64
summingInt collector, 64
super syntax, 53
supplyAsync method (CompletableFuture), 151
Swing, 5

T

take method (Observable), 154
target types of lambda expressions, 7, 46
task parallelism, 82
template method pattern, 119–122
  lambda expressions vs., 120
test doubles, 105–106
Test-Driven Development (Beck), 97
test-driven development (TDD), 97
  behavior-driven development vs., 123
testing
  concurrent systems with verticals, 145
  logging, 106
  midstream breakpoints, 107

verticals, 145
thenAccept method (CompletableFutures), 152
thenApply method (CompletableFutures), 152
thenCombine method (CompletableFuture), 150
thenRun method (CompletableFutures), 152
ThreadLocal class, 132
lonely overrides and, 98
toCollection collector, 62
toList collector, 62
ToLongFunction method, 43
toSet collector, 62
TreeSet, 63
in parallel processing, 91
Twitter, 155
type inference, 12
types
inferring, 11–13
parameter, 9
Predicates and, 12
primitive, 42–45

U

unboxing, 43
unit testing, 102–104
challenges of, 102
test doubles and, 105–106
unordered method (Stream), 62
user connections, handling, 141
utility code, 54

V

values
final, 8
variables vs., 8
variables vs. values, 8
verbosity of code, 2
vertical (vertx), 140
vertx framework, 139
broadcast messages, implementing within, 143
Buffer class, 144
CompletableFutures in, 151
connectHandler method, 141
dataHandler method, 141
eventbus, 142, 143
implementing code within, 140
integrating with RxJava, 155
Message object, 142
verticals, 140
virtual methods, 50

W

with method, 146
Write Everything Twice (WET), 99–102

Z

zip algorithm, 114

About the Author

Richard Warburton is an empirical technologist and solver of deep-dive technical problems. He has professionally worked on static analysis problems, verifying part of a compiler and developing advanced automated bug detection technology. More recently his career has been focused on data analytics for high-performance computing. He is a leader in the London Java Community, sits on its JCP Committee, and organizes the Adopt-a-JSR programs for Lambdas and Date and Time in Java 8. Richard is also a known conference speaker, having talked at JavaOne, DevoxxUK, and JAX London. He obtained a PhD in Computer Science from the University of Warwick, where his research focused on compiler theory.

Colophon

The animal on the cover of *Java 8 Lambdas* is a lesser spotted eagle (*Aquila pomarina*). This large bird of prey can be found in Eastern Europe and belongs to the family Accipitridae, like all typical eagles. The lesser spotted eagle is medium-sized with a head and bill that are small for eagles; these eagles typically measure up to 60 cm in length with a 150 cm wingspan.

Juvenile spotted eagles have white spots on their flight feathers, also called remiges, whereas adults are pale brown in the head and wings with dark plumage. These eagles breed in central and eastern Europe, laying 1–3 white buff-spotted eggs in a tree nest. As is typical for eagles, the number of young depends on the amount of food during the breeding season. The female begins incubation when the first egg is laid; often the first young outgrows its clutchmate and eventually kills or eats them.

The cover image is from Meyers Kleines. The cover fonts are URW Typewriter 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.