O'REILLY®

A Functional Approach to Java

Augmenting Object-Oriented Java Code with Functional Principles

Early Release
RAW & UNEDITED

Ben Weidig

A Functional Approach to Java

Augmenting Object-Oriented Code with Functional Principles

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Ben Weidig

A Functional Approach to Java

by Ben Weidig

Printed in the United States of America.

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

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

- Editors: Rita Fernando and Suzanne McQuade
- Production Editor: Ashley Stussy
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- September 2022: First Edition

Revision History for the Early Release

- 2021-08-26: First Release
- 2021-01-18: Second Release
- 2021-03-29: Third Release
- 2022-05-17: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098109929 for release details.

978-1-098-10985-1

Preface

A mind that is stretched by a new experience can never go back to its old dimensions.

—Oliver Wendell Holmes Jr.

Developing software can be quite a complex endeavor. As Java developers, we usually try to tame this complexity with object-oriented programming (OOP) and an imperative coding style. But not every problem is a good match for an object-oriented coding style. We end up introducing even more complexity by not solving problems with the best tools and paradigms available to us. The functional programming (FP) paradigm offers *another* approach to solving problems.

After spending its early life hidden away in academia and niches, functional programming is on the rise again and becoming more mainstream. The ideas and concepts behind it are adopted in almost every multi-paradigm and general-purpose language, allowing us to use some form of functional programming regardless of the context. And it's not a surprising trend.

New Hardware Needs a New Way of Thinking

Our hardware is evolving in a new direction. *Moore's law* — coined in 1965 as the observation of transistor counts doubling every two years, and therefore the performance per core available to us — seems to slow down.[1] For quite some time, single-core performance improvements are getting smaller with each processor generation. The manufacturers favor more cores — even specialized ones — over ever-increasing transistor count and clock rates per core.[2] That's why modern workloads need new ways to reap all the benefits such new hardware offers: parallelism.

CONCURRENCY VERSUS PARALLELISM

Concurrency and parallelism are often confused with each other or seen as the same thing. The Merriam-Webster dictionary even defines them quite similar:

Concurrence

> The simultaneous occurrence of events or circumstances.

Parallel

> An arrangement or state that permits several operations or tasks to be performed simultaneously rather than consecutively.

But in computer science, the terms express two different concepts.

Concurrency allows multiple threads to make progress simultaneously on the same CPU core. The threads need to be coordinated and “interrupted” to get their work done. Think of it like a juggler using only one hand (single CPU core) with multiple balls (threads). They can only hold a single ball at any time (doing the work), but which ball changes over time (interrupting and switching to another thread). Even with only two balls, they have to juggle the workload.

Parallelism is about running multiple tasks at literally the same time, like on multiple CPU cores. The juggler now uses both hands (more than one CPU core) to hold two balls at once (doing the work simultaneously). If there are only two balls in total, they can hold both at the same time.

These concepts aren’t mutually exclusive and are often used together.

Scaling your software *horizontally* through parallelism isn’t an easy task in OOP. Not every problem is a good fit for parallelism. More painters might paint a room faster, but you can’t speed up pregnancy by involving more people. If the problem consists of serial or interdependent tasks, concurrency is preferable to parallelism.

But if a problem can be broken down into smaller, non-related sub-problems, parallelism really shines. And the stateless and immutable nature of idiomatic FP provides all the tools necessary to build small, reusable tasks to be easily used in parallel environments. That's just one of many benefits of a more functional approach to your daily development problems.

Why Java?

There are many programming languages to choose from when you want to start with functional programming. *Haskell* is a favorite if you prefer a *pure* functional language with almost no support for an imperative coding style. But you don't have to leave the JVM ecosystem behind to find FP-capable languages. *Scala* shines in combining OOP and FP paradigms into a concise, high-level language. Another popular choice, *Clojure*, was designed as a functional language with a dynamic type system at heart. But sometimes, you won't have the luxury of choosing the language for your project or problem, and you'll have to play the cards you're dealt, and you'll need to use Java.

Even though you can implement most functional principles in Java regardless of deeply integrated language level support[3], your code won't be as concise and easy to reason with as it would in other languages. And without such support, many developers didn't bother to embrace these principles, even if they could have provided a more productive approach or better overall solution.

In the past, many people thought of Java as a slow-moving behemoth, a "too big to become extinct" enterprise language, like a more modern version of COBOL or FORTRAN. And in my opinion, that's partially true. The pace didn't pick up until Java 9 and the shortened release timeframes[4]. It took Java five years to go from version 6 to 7 (2006-2011). And even though there were significant new features, like `try-with-resources`, none of them were "ground-breaking". The few and slow changes in the past led to projects and developers not adopting the "latest and greatest" Java Development Kit (JDK), missing out on many language improvements. Three years later, in 2014, the next version, Java 8, was released. But this time, it introduced one of the most significant changes to Java's future: *lambda expressions*.

A better foundation for functional programming had finally arrived in arguably the most prominent object-oriented programming language of the world, changing the language and its idioms significantly:

```
Runnable runnable = () -> System.out.println("hello, functional
world!");
```

A whole new world of ideas and concepts was made available to Java developers by introducing lambda expressions. Many of the JDK's new features, like Streams, or the `Optional` type, are only possible in such a concise way thanks to language-level lambda expressions. But the new idioms and way of doing things with FP might not come naturally, especially after spending so much time in an "object-oriented headspace".

Why I Wrote This Book

After using a more functional style in other languages I work with, like *Swift*, I gradually introduced more functional principles in my Java-based projects. That led me to realize something: *How* to use lambdas, Streams, and other functional tools provided by Java, is easy to grasp. But without understanding *why* you should use them — and when not to — you won't unlock their full potential, and it will just be "new wine in old wineskins."

So I decided to write this book to highlight the different concepts that make a language *functional*, and how you can incorporate them into your Java code, either with the tools provided by the JDK or by creating them yourself. A functional approach to your Java code will most likely challenge the status quo and go against *best practices* you were using before. But by embracing a more functional way of doing things, like *immutability* and *pure functions*, you will be able to write more concise, more reasonable, and future-proof code that is less prone to bugs.

Who Should Read This Book

This book is for you if you are curious about functional programming and want to know what all the fuss is about and apply it to your Java code. You might already be using some functional Java types but desire a more profound knowledge of why and how to apply them more effectively.

There is no need to be an expert on OOP, but the book is not a beginner's guide to Java or OOP. You should already be familiar with the Java standard library. No prior knowledge of functional programming is required. Every concept is introduced with an explanation and examples.

The book covers Java 17 as the latest Long-Term-Support (LTS) version available at publication. But knowing that many developers need to support projects with earlier versions, the baseline will be the previous LTS, Java 11.

This book might not be for you if you are looking for a compartmentalized, recipe-style book presenting "ready-to-implement" solutions. Its main intention is to introduce functional concepts and idioms and teach you how to incorporate them into your Java code.

What You Will Learn

By the end of this book, you will have a fundamental knowledge of functional programming and its underlying concepts and how to apply this knowledge to your daily work. Every Java functional type will be at your disposal, and you will be able to build anything missing from the JDK by yourself, if necessary.

A functional approach will lead to many advantages in your code:

- *Composition*: Build modular and easy composable blocks.
- *Expressiveness*: Write more concise code that clearly expresses its intent.
- *More reasonable code*: Safer data structures without side-effects that don't need to deal with locks or race conditions.
- *Modularity*: Break down larger projects into more easily manageable modules.
- *Maintainability*: Smaller functional blocks with less interconnection make changes and refactoring safer without breaking other parts of your code.
- *Data manipulation*: Build efficient data manipulation pipelines with less complexity.
- *Performance*: Immutability and predictability allow to scale horizontally with parallelism without much thought about it.
- *Testing*: Verify your building blocks with ease.

Even without going *fully functional*, your code will benefit from the concepts and idioms presented in this book. And not only your Java code. You will tackle development challenges with a functional mindset, improving your programming regardless of the used language or paradigm.

What About Android?

Talking about Android in a Java context is always a challenging endeavor. Even though you can write Android applications in Java, the underlying API isn't the same, and Android doesn't run Java bytecode on a JVM. Instead, it recompiles the Java bytecode for its own runtime.

Android is (not) Java

Android chose Java as its primary language for multiple reasons. At the time of Android's inception, Java was a well-known language and the first programming language many universities taught their students. Also, it offered a vast pool of developers and a vibrant ecosystem of compatible libraries. But instead of running Java bytecode on a minimalistic JVM, like Java Platform Micro Edition, the Java bytecode is recompiled. The *Dex-compiler* creates *Dalvik bytecode*, which runs on a specialized runtime: the *Android Runtime* (ART), and previously on the *Dalvik virtual machine* [5].

Recompiling Java bytecode to *Dalvik bytecode* allows the devices to run highly optimized code, getting the most out of their hardware constraints. But for you as a developer, that means that even though your code looks and feels like Java, and most of the public API is available to you, there isn't a feature parity between the JDK and Android SDK you can rely on. For example, the cornerstones of this book — *lambda expressions* and *streams* — were among the missing features in Android for a long time.

Desugaring Android Java Code

The expression "syntactic sugar" describes features that are additions to a language's syntax to make your life as a developer "sweeter". It provides an alternative, more concise style for more complex tasks. You will learn more about in "Syntactic Sugar". For example, augmented assignments, prefix and postfix operators, and type inference, as shown in Table P-1, are "syntactic sugar" you might already use.

Table P-1. Syntactic sugar

Description	With Sugar	Without Sugar
Type inference	var x = 42L;	long x = 42L;
Type inference	List<String> list = new ArrayList<>();	List<String> list = new ArrayList<String>();
Augmented Assignment	x += 1;	x = x + 1;
Postfix Operator	x++;	x = x + 1;

The compiler is responsible for removing the “sweetness” by *desugaring* your code, returning it to the actual form that gets compiled.

Java lambda expressions are more than just “syntactic sugar”, as you will learn more about in “Lambdas Versus Anonymous Classes”. But for Android, there was no other option to support various Java 8+ features than *desugaring*. At least without implementing them natively at a runtime level. Starting with 3.0.0, the Android Gradle plugin supports automatic *desugaring* of the following features that are covered in this book:

- Lambda expressions (without serialization support)
- Method references
- Default and static interface methods

The next major version, 4.0.0, added even more functional features:

- Streams
- Optionals
- The `java.util.function` package

Keep in mind that even though all these features are finally available in Android, they are implemented differently from the JDK[6].

A Functional Approach to Android

In 2019, Google announced that Java is no longer the preferred language for Android app developers. It got replaced by Kotlin, after making it an available option two years prior. Kotlin is a multi-platform language that mainly targets the JVM but also compiles to JavaScript and many multiple native platforms, too[7]. It aims to be a “modern and more concise” Java, fixing many of Java’s *debatable* shortcomings and cruft accumulated over the years due to backward compatibility, without forgoing all the available frameworks and libraries available in Java. It’s 100% interoperable with Java, and you can mix Java and Kotlin in the same project with ease.

One obvious advantage of Kotlin over Java is that many of the introduced functional concepts and idioms are an integral part of the language itself. But Kotlin has its own idioms and best practices that differ from Java's. The generated bytecode might differ, too, like how to generate lambdas[8]. The most significant advantage of Kotlin is its attempt to create a more concise and predictable language compared to Java. And just like you can be more functional in Java without going *fully functional*, you can use Kotlin-only features without going *full Kotlin* in your Android projects. By mixing Java and Kotlin, you can pick the best features from both sides.

Keep in mind that this book's primary focus is Java. Most of the ideas behind what you will learn are transferrable to Android, even if you use Kotlin. But there won't be any special considerations for Android throughout the book.

Navigating This Book

This book consists of three different parts. Reading them in their respective order will let you get the most of them because they build on each other. The contained chapters, however, are only loosely coupled. So feel free to skim for the bits that might interest you and jump around. Any necessary connections are cross-referenced.

- Part I, *A Functional Approach*, covers a high-level overview of functional programming and the types already available to Java developers to better understand the different concepts' underlying philosophy. It's followed by a topic-based deep-dive through the different concepts and how to use them.
- In [Link to Come], *Real-World Problems, Patterns and Recipes*, you will see how to apply the previously learned knowledge to typical *real-world problems* you might encounter in your daily work.

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.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

The source code for the book is available on GitHub: *https://github.com/benweidig/a-functional-approach-to-java*. Besides compilable Java code, there are also *JShell* scripts available to run the code more easily. See the README.md for instructions on how to use them.

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

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

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

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

O’Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

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 *https://learning.oreilly.com/home/*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://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

---

1 Edwards, Chris. 2021. "Moore's Law: What Comes Next?" Communications of the ACM, February 2021, Vol. 64 No. 2, 12–14.

2 Thompson, N. C., and Svenja Spanuth. 2021. "The decline of computers as a general-purpose technology." *Communications of the ACM*, Vol. 64, No. 3, 64-72.

3 Dean Wampler shows in his book "Functional Programming for Java Developers" quite detailed how to implement and facilitate the missing functional programming features in Java all by yourself. He showed many techniques that weren't easily feasible before version 8. But now, many of the shortcomings and gaps in the JDK are closed up, and it provides many of the tools necessary to incorporate FP concisely and more straightforward.

4 Oracle introduced a faster release schedule for Java with the release of version 9. Instead of releasing infrequently, there's now a fixed release cadence of six months. To meet such a tight schedule, not every release is considered "long-term-support", in favor of releasing features faster than before.

5 The Android Open Source project provides a good overview of the features and the reasoning behind Android's runtime.

6 Jack Wharton, a well-known Android developer, provides a detailed insight on how Android desugars modern Java code.

7 See the official Kotlin documentation for an overview of supported platforms.

8 Each lambda compiles to an anonymous class extending `kotlin.jvm.internal.FunctionImpl`, as explained in the function type specs.

Part I. A Functional Approach

Functional programming isn't more complicated than object-oriented programming and its primarily imperative coding style. It's just a different way of approaching the same problems. Every problem that you can solve imperatively can also be solved functionally.

Mathematics builds the foundation for functional programming, making it harder to approach than an object-oriented mindset. But just like learning a new foreign language, the similarities and shared roots become more visible over time until it *just clicks*.

You can implement almost any of the upcoming concepts without Java lambda expression. But compared to other languages, the result won't be as elegant and concise. The functional tools available in Java allow your implementations of these concepts and functional idioms to be less verbose and more concise and efficient.

Chapter 1. An Introduction to Functional Programming

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

To better understand how to incorporate a more functional programming style in Java, we first need to make ourselves knowledgeable about functional programming's origin and foundational concepts. This chapter will explore the roots of functional programming and what concepts contribute to making an approach to programming more functional.

The Origin of Functional Programming

The functional programming paradigm evolved from *Lambda Calculus*, invented by the logician Alonzo Church in the 1930s.[1] Lambda calculus is a formal mathematical system to express computations with abstract functions and apply variables to them. The name "lambda calculus" came from the Greek letter chosen for its symbol: λ

Don't worry! I promise I won't torture you with more complex math than needed.

There are three pillars to support the general concept of lambda calculus:

- *Abstraction*
- *Application*
- *Reduction*

Lambda Abstractions

What you as a developer refer to as a "function call" is, mathematically speaking, the application of *lambda abstraction* to a value. It can be declared as a function like this: $f = \lambda x.E$

A function declaration consists of multiple parts:

x

A *variable*, the argument representing a value.

E

An *expression*, or *term*, containing the logic.

$\lambda x.E$

The *abstraction*, an anonymous function accepting a single input x.

f

The resulting function that can apply an argument to its abstraction.

Let's imagine a Java function for calculation a quadratic value, as seen in Example 1-1.

Example 1-1. Quadratic function (Java)

```
// As "classical" method

Integer f(Integer value) {
```

```
  return value * value;
}

// As lambda expression

Function<Integer, Integer> f = x -> x * x; ❶
```

❶ A functiona accepting a single `Integer`, and returning an `Integer`.

> **NOTE**
>
> Example 1-1 uses `Integer` instead of `int` due to generic nature of Java's functional types. The use of value types in generics is part of the upcoming Project Valhalla.

The Java lambda expression quite resembles its lambda calculus counterpart in Equation 1-1.

Equation 1-1. Quadratic function (lambda calculus)

$$f = \lambda x.\, x^* x$$

Application

The application of an abstraction in Equation 1-2 looks like a method call that you're quite used to in Java.

Equation 1-2. Application of f to argument 5 (lambda calculus)

$$f5$$

The Java equivalent in Example 1-2 is a little bit more verbose because it uses the "normal" method calling syntax that requires a name, but you can't deny the similarity.

Example 1-2. Application of f to argument 5 (lambda calculus)

```
// As applied lambda expression

f.apply(5);
```

```
// As method call

f(5);
```

Reduction

If you apply a lambda abstraction to an argument, the variable in the expression gets substituted by the argument. This form of substitution is called *β-reduction*, as seen in Equation 1-3.

Equation 1-3. β-reduction

$$\begin{aligned} & f5 \\ \rightarrow \quad & 5*5 \\ \rightarrow \quad & 25 \end{aligned}$$

$$\begin{aligned} & f(f3) \\ \rightarrow \quad & f(3*3) \\ \rightarrow \quad & f9 \\ \rightarrow \quad & 9*9 \\ \rightarrow \quad & 81 \end{aligned}$$

The equivalency between a function application and the result itself allows simplifying more complex constructs. Complex calculations will be more approachable and less intimidating after reducing them to a more simple form.

Of course, there are way more details to it[2], but that's all you will need to understand the origin of functional programming.

LAMBDA CALCULUS AND TURING MACHINES

Another computational model developed in the 1930s[3] is the "Turing machine". Its inventor — Alan Turing — was a Ph.D. student of Alonzo Church at Princeton 1936-1938, but they developed their respective computational models independently before working together.

The stateless *lambda calculus* seems to be a contrary approach to mathematical calculations than to *turing machines* and their internal state. But they are both models that capture the notion of algorithmic computation. And it turned out that *lambda calculus* [4] is actually *turing complete*, meaning it can calculate anything a Turing machine could calculate, creating an unintended equivalency between the two models.[5]

What is Functional Programming?

Like most paradigms, functional programming doesn't have a single agreed-upon definition, and many turf wars are fought about what defines a language as *really* functional. Instead of giving my own definition, I will show you different aspects of what makes a language functional.

As an object-oriented developer, you are used to *imperative* programming: by defining a series of statements, you are telling the computer *what* to do to accomplish a particular task.

Functional programming uses a *declarative* style to express the logic of computations without describing their control flow. It is a description of *how* a program should work, not *what* it should do. Your code is bound in a sequence of functions, representing evaluable *expressions* instead of *statements*.

The primary distinction between *expressions* and *statements* is that the latter has possible *side-effects* to program state, and the former is supposed to be self-contained with *immutable* state. These properties aren't absolute or mutually exclusive. Especially in a general-purpose, multi-paradigm language like Java, the lines between them can quickly blur.

Expressions

An *expression* is a sequence of operators and operands that define a computation, like in Example 1-3. An expression *can* return some form of a result but doesn't have to. They are analogous to the concept shown in "Lambda Abstractions". Side-effects are discouraged but aren't forbidden either.

Example 1-3. Simple Java Expressions

```
x * x ❶

2 * Math.PI * radius ❷
```

❶ The quadratic expression used in Example 1-1.
❷ An expression to calculate the circumference of a circle.

Statements

In Java, you're used to statements. Assigning or changing the value of a variable, calling methods, or control-flow like `if/else`; all of these are statements. They are *actions* taken by your code, as in Example 1-4.

Example 1-4. Java Statements

```
int treasureCounter = 0; ❶

treasureCounter += findTreasure(6); ❷

if (treasureCounter > 10) { ❸
  System.out.println("You have a lot of treasure!");
}
else {
  System.out.println("You should look for more treasure!");
}
```

❶ Assigns an initial value to a variable, introducing state into the program.
❷ The function call `findTreasure(6)` might be a pure functional expression, but the reassignment of `treasureCounter` is state-change and therefore a statement.

❸ The control-flow statement `if/else` expresses what action should be taken based on the result of the expression `(treasureCounter > 10)`.

Functional Programming Concepts

Functional programming is a conglomerate of different concepts, forming a paradigm in which everything is bound together with pure mathematical functions. Its primary focus is on “what to solve” in a declarative style, in contrast to the imperative “how to solve” approach.

We will go through the most common and significant aspects functional programming builds upon. But remember, these aren’t exclusive to a particular paradigm. Many of the ideas behind them apply to other programming paradigms as well.

Pure Functions

Functional programming categorizes functions into two categories: *pure* and *impure*.

Pure functions have two elemental guarantees:

- The *same* input will *always* create the same output.
- They are self-contained without any kind of side-effect, e.g., affecting the global state or changing argument values, or using I/O, like in Figure 1-1.

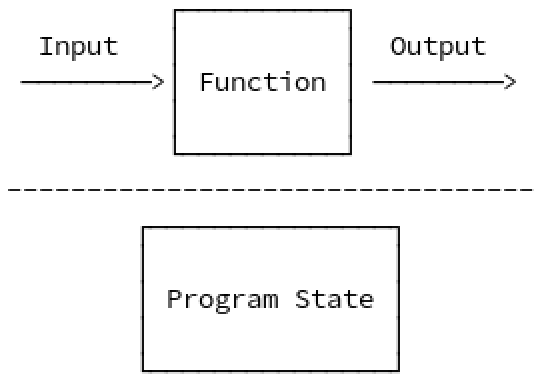

Figure 1-1. Pure Functions are separated from Program State

These two guarantees allow *pure functions* to be safe to use in any environment, even in a parallel fashion.

Functions violating any of these guarantees are considered *impure*. That is a rather unfortunate name because of the connotation it might invoke. *Impure functions* aren't second-class to *pure functions*. They are just used in different ways.

Referential Transparency

Due to the predictable result of side-effect-free *expressions* and *pure functions* based on their input, their respective return values can replace them for any further invocations once evaluated, without changing the result of the program. These kinds of functions and expressions are *referentially transparent*. You have seen this kind of substitution in Equation 1-3.

Optimization techniques, like *memoization*, can use this concept to cache function calls to prevent unnecessary reevaluation of expressions.

Immutability

Object-oriented code, like in Java, is often based around a mutual state. Objects can usually be changed after their creation, using *setters*. But mutating data structures can create unexpected side effects.

With *immutability*, data structures can no longer change after their initialization. By never changing, they are always consistent, and therefore predictable, side-effect-free, and easier to reason with. Like *pure functions*, their usage is safe in concurrent and parallel environments without the usual issues of unsynchronized access or out-of-scope state changes.

If data structures never change at all, a program would not be very useful. Instead of mutating existing data, you have to create a new data structure containing the changed data. At first, this might sound like a chore, and actually, it can be. But in general, the advantages of having side-effect-free data structures outweigh the extra work that might be necessary.

Recursion

Recursion is an approach for problems that can be partially solved, with a remaining problem in the same form. In layman's terms, recursive functions call themselves, but with a slight change in their input arguments until they reach an end condition and return an actual value. The later chapter "Mathematical Explanation" will go into the more finer details of recursion.

A simple example is calculating a factorial, the product of all positive integers less than or equal to the input parameter. Instead of calculating the value with an intermediate state, the function calls itself with a decremented input variable, like in Figure 1-2.

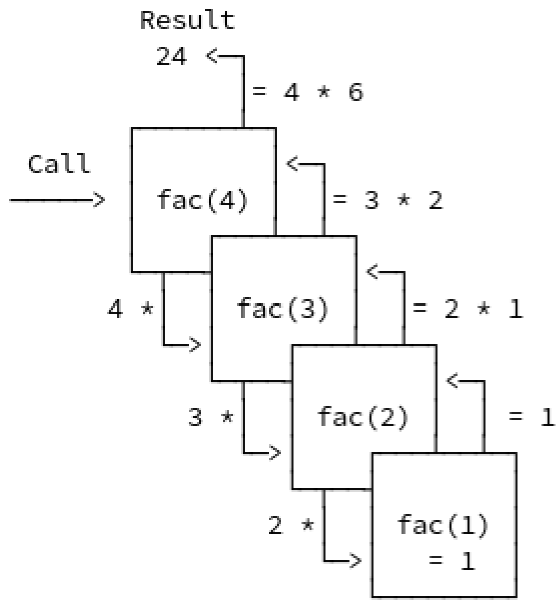

Figure 1-2. Calculating a factorial with recursion

Pure functional programming prefers using recursion instead of loops. Some languages, like Haskell, don't even provide the *traditional* `for` or `while`-loops.

As with most of the other concepts, *recursion* is also not exclusive to functional programming.

First-Class and Higher-Order

Many of the previous concepts don't have to be (fully) available to support a more functional programming style in a language. But this one is an absolute must-have.

Functions are supposed to be a "first-class citizen", giving them all the properties inherent to other entities of the language. They need to be assignable to variables and be used as arguments and return values in other functions and expressions, like in Example 1-5.

Example 1-5. First-Class Functions

```
Function<Integer, Integer> quadraticFn = x -> x * x; ❶

var result = quadraticFn.apply(5); ❷
```

Expressions are based on so-called *functional interfaces*, and can be

❶ Expressions are based on so-called *functional interfaces*, and can be assigned to variables like any other value.

❷ It can be used like any other “normal” Java variable, calling the `apply` method of its interface.

Higher-order functions use their *first-class citizenship* to accept functions as arguments or to return a function as their result, or both. That is essential for the next concept, *functional composition*.

Functional Composition

Pure functions can be combined to create more complex expressions. In mathematical terms, this means that the two functions $f(x)$ and $g(x)$ can be combined to a function $h(x) = g(f(x))$, as seen in Figure 1-3.

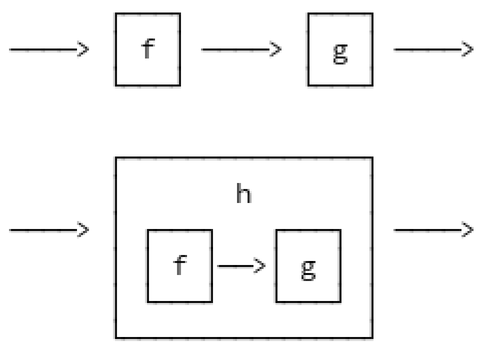

Figure 1-3. Composing functions

This way, the initial functions can be small and reusable, and the resulting composed function will perform a more complex and complete task. Let’s combine the previous quadratic function with other functions in Example 1-6.

Example 1-6. Functional Composition in Java

```
Function<Integer, Integer> quadratic = x -> x * x; ❶

Function<Integer, Integer> triple = x -> 3 * x; ❷

var quadraticThenTriple = quadratic.andThen(triple); ❸

var tripleThenQuadratic = quadratic.compose(triple); ❹
```

```
var result1 = quadraticThenTriple.apply(3); // => 27

var result2 = tripleThenQuadratic.apply(3); // => 81
```

❶ The simple quadratic function from previous examples.
❷ Another pure function, tripling the applied value.
❸ Composing a function, calling `triple` with the result of `quadratic`.
❹ Composing a function the other way around, tripling first.

If you look at the source code of `andThen` and `compose`, you can clearly see the concept of *first-class* and *higher-order* functions in action. Example 1-7 is a simplified version of what's actually happening.

Example 1-7. Source code of `andThen` and `compose`.

```
Function<Integer> andThen(Function<Integer, Integer> after) { ❶
  return value -> after.apply(apply(value)); ❷
}

Function<Integer, Integer> compose(Function<Integer, Integer>
before) { ❸
  return value -> apply(before.apply(value)); ❹
}
```

❶ `andThen` is a *higher-order* function, accepting a function as its argument, and returning a combined function.
❷ The returned function accepts a value and applies the current function to it first, and the result is then applied to `after`.
❸ As with `andThen`, a function is accepted as an argument and returns a new function.
❹ This time, `before` is applied to the value first, and the original function is applied to the result.

Laziness

Lazy evaluation is a common technique to decouple the evaluation of an expression until its result is actually needed. Expressions evaluate *just-in-time*. It is another concept that is not rooted in functional programming itself but provides a foundation for many related concepts.

Some form of laziness is already available in Java: logical short-circuit operators, as seen in Example 1-8.

Example 1-8. Logical Short-Circuit Operators

```
var result1 = simple() && complex();

var result2 = simple() || complex();
```

The number of evaluated expressions depends on the results of `simple()`, as seen in Table 1-1.

Table 1-1. Evaluation of Logical Operators

Operator	Result of `simple()`	Is `complex()` evaluated?
`&&`	`true`	yes
`&&`	`false`	no
`||`	`true`	no
`||`	`false`	yes

The JVM can discard expressions not related to the final result. This behavior even allows infinite data structures to exist, as you will learn about in [Link to Come].

Laziness works well with *referential transparency*. If there is no difference between an expression and its result, it doesn't matter when you will execute it. Delayed evaluation might still impact the program performance because you might not know the precise time of evaluation.

Advantages of Functional Programming

Now that you have learned about the different concepts functional programming relies on, what advantages does it provide to you and your code?

Simplicity

: Without state and side-effects, your functions tend to be smaller, doing "just what they are supposed to do".

Consistency

: Immutable data structures are reliable and consistent. No more worries about changed state without you knowing.

(Mathematical) Correctness

Simpler code with consistent data structures will automatically lead to "more correct" code with fewer bugs. The "purer" your code, the easier it will be to reason with, leading to easier debugging and testing.

Concurrency

Concurrency is one of the most challenging tasks to do right in "classical" Java. Functional concepts allow you to eliminate many headaches and gain safer parallel-processing (almost) for free.

Modularity

Small, independent, and reusable functions allow a new form of modularity and reusability, like functional composition.

Academia Versus "The Real World"

The foundation of functional concepts consists of strictly mathematical principles due to their roots in academia. That provides us with a straightforward, easy to reason with, and safe paradigm.

But all of us know that not everything obeys the rules, especially in *real-world* projects. That's why many functional programming languages deviate from the *purest* interpretation of the fundamental concepts for various reasons, most likely to provide a broader range of use. And even then, how much of the remaining strictness you want to introduce into your code relies mostly on you and your requirements.

Language design is always a balancing act between *being safe* and *being convenient*. For example, Haskell, a purely functional programming language, has the slogan "avoid success at all costs". "Success" in this case means broad popularity and widespread use, and "costs" being concessions made to further such "success". They won't "make things easier" for beginners or add any changes that might impact the core values of Haskell. That can make a language "useless" but "safe". Simon Peyton Jones, lead developer of the Glasgow Haskell Compiler and major contributor to

Haskell, describes the relationship between the two properties in an informal Youtube video.

With Haskell arguably being an academia language with only niche-adaption, it can afford to stand up for its convictions. Java does it, too, but has other priorities. Every new Java version provides you with safer and more useful tools.

The goal you should strive for in your own code shouldn't be relying on one extreme position or another. Instead, it should be the amalgamation of the best of both worlds.

Takeaways

- The mathematical principle of lambda calculus and abstractions builds the foundation for FP.
- FP emphasizes expressions, while imperative programming emphasizes statements.
- There are many inherently functional concepts, but they are not an absolute requirement to make code "functional".
- Trade-offs are often necessary between "pureness" of functional concepts and their real-world application.

---

1 Church, Alonzo. 1936. "An unsolvable problem of elementary number theory". American journal of mathematics, Vol. 58, 345–363.

2 The Wikipedia entry on lambda calculus provides more information.

3 Turing, A.M. 1937. "On Computable Numbers, with an Application to the Entscheidungsproblem." Proceedings of the London Mathematical Society, Vol. s2-42 Issue 1, 230-265.

4 There are certain restrictions on *lambda calculus* to be *turing complete*. See the Wikipedia entry on Turing completeness for more information.

5 Copeland, Jack and Oron Shagrir. 2019. "The Church-Turing Thesis: Logical Limit or Breachable Barrier?" Communications of the ACM, January 2019, Vol. 62 No. 1, Pages 66-74.

Chapter 2. Functions and Lambdas

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

Unsurprisingly, *lambdas* are the key to functional programming. But how do lambdas in Java exactly work? And how can you use them to incorporate functional concepts in your code?

In this chapter, you will learn about what Java lambdas are and how they work internally. You will get to know the functional types available in the JDK and the different concepts and ideas they enable you to implement in your code.

Java Lambdas

Objects and primitives in Java are "first-class citizens." They can be freely assigned to variables and passed into and returned from methods. Functional programming requires functions to be "first-class citizens." Without it, many concepts and techniques aren't possible or heavily restricted. But Java is based on methods bound to objects, not standalone functions.

From a simplified point of view, a lambda is like an *anonymous method* that doesn’t belong to any object. It still behaves like any other object or variable in Java: it has a particular type, it’s assignable to a variable, usable as an argument, etc. But lambdas are concrete implementations of so-called *functional interfaces*, which have certain constraints that differ from what you might be used to.

Functional Interfaces

There isn’t any explicit syntax or keyword for *functional interfaces*. They look and feel like any other interface, can extend or be extended by other interfaces, and classes can implement them. So if they are just like “normal” interfaces, what makes them a *functional interface* then? It’s their requirement that they may only define a *single abstract method* (SAM).

INTERFACES IN JAVA

Interface declarations consist of a name with optional generic bounds, inherited interfaces, and its body. Such a body is allowed to contain the following content:

Method signatures

> Body-less — `abstract` — method signatures that must be implemented by any class conforming to the interface. Only these method signatures count towards the *single abstract method* constraint of *functional interfaces*.

`default` *methods*

> Methods signatures with a “default” implementation, signified by the `default` keyword. Any class implementing the interface *can* override it but *isn’t required* to do so.

`static` *methods*

> Like the class-based counterparts, they’re associated with the type itself and must provide an implementation. But unlike `default` methods, they aren’t inherited and can’t be overridden.

Constant values

> Values that are automatically `public`, `static`, and `final`.

As the name signifies, this restriction applies only to `abstract` methods. There’s no limit to any additional, non-`abstract` methods. Neither `default` nor `static` methods are abstract, hence not relevant for the SAM count. They are often used to complement the capabilities of the lambda type.

For example, the type `java.util.function.Predicate<T>` is a functional interface. Besides it SAM — `boolean test(T t)` —, it provides five additional methods (three `default`, two `static`), as you can see in the simplified interface[1] declaration in Example 2-1.

Example 2-1. Simplified java.util.functional.Predicate<T>

```
package java.util.function;

@FunctionalInterface ❶
public interface Predicate<T> {

  boolean test(T t); ❷

  default Predicate<T> and(Predicate<? super T> other) { ❸
    // ...
  }

  default Predicate<T> negate() { ❸
    // ...
  }

  default Predicate<T> or(Predicate<? super T> other) { ❸
    // ...
  }

  static <T> Predicate<T> isEqual(Object targetRef) { ❹
    // ...
  }

  static <T> Predicate<T> not(Predicate<? super T> target) { ❹
    // ...
  }
}
```

❶ The type has a `@FunctionalInterface` annotation, which isn't explicitly required.

❷ The *single abstract method* of the type `Predicate<T>`.

❸ Several `default` methods provides support for functional composition.

❹ Convenience `static` methods are used to simplify creation or to wrap existing lambdas.

Even though the *single abstract method* requirement is the *only* requirement a functional interface has to oblige to, all JDK-provided functional interfaces use the explicit `@FunctionalInterface` annotation. It isn't mandatory, and it doesn't provide any other functionality than marking a type as a functional interface. Its purpose is to tell the compiler or any tooling that works with annotations that a type is supposed to be a functional interface and that the *single abstract method* requirement must be enforced. If you add another `abstract` method, the Java compiler will refuse to compile your code. That's why adding the annotation makes a lot of sense, even if you don't explicitly need it. It makes the reasoning and intention of an interface clearer and fortifies your code against unintentional changes that might break it in the future.

Generic Signatures

TODO

Lambda Syntax

In "Lambda Abstractions" you've already learned about the mathematical notation for lambdas:

$\lambda x . E$

The actual Java syntax doesn't differ that much, as shown in Example 2-2.

Example 2-2. Java lambda syntax

```
(<parameters>) -> { <body> };
```

A Java lambda consists of three distinct parts:

Parameters

A comma-separated list of parameters, just like a method argument list. You can omit the type completely if the compiler can infer them. In case of multiple parameters, you must wrap them in parenthesis. But for a single parameter, they are optional. Mixing implicitly and explicitly typed parameters is not allowed.

Arrow

The -> (arrow) separates the parameters from the lambda body. It's the equivalent to λ in lambda calculus.

Body

The body is either a single expression or a block statement. The curly braces aren't allowed for single expressions, and the evaluated result returns implicitly without a `return` statement. But if the body is represented by more than a single expression, a typical Java code block is used instead. It must be wrapped in curly braces and explicitly use a `return` statement if the functional interface requires to return a value.

That is all the syntax definition there is for lambdas in Java. With its multiple ways of declaring a lambda, you can write the same lambda with different verbosity levels, as seen in Example 2-3.

Example 2-3. Different ways of writing the same lambda

```
Predicate<String> isNotNull = (String input) -> { ❶
  return input != null;
};

Predicate<String> isNotNull = input -> { ❷
  return input != null;
};

Predicate<String> isNotNull = input -> input != null; ❸
```

❶ The most verbose variant: an explicitly typed parameter in parenthesis and the body as a block.

❷ Type inference for parameters allows removing the explicit type, and a single parameter doesn't need parenthesis. That shortens the lambda declaration slightly without removing information due to the surrounding context.

❸ Reducing the body to a single expression allows you to remove the curly braces and the need for the `return` keyword.

Which variant to choose depends highly on the context and personal preference. Usually, the compiler can infer the types and deduce any missing information. But that doesn't mean a human reader is as good at understanding the shortest code possible, just like a compiler. Even though you should always strive for clean and more concise code, that doesn't mean it has to be as minimal as possible. A certain amount of verbosity might help understand the reasoning behind your code better and make it fit into the mental model of your code more efficiently.

Calling Lambdas

With lambdas effectively being concrete implementations of their respective functional interfaces, their usage differs from other, more functional languages. For example, JavaScript, or even JVM languages like Scala, allow you to call a lambda directly. But Java decided to implement functional interfaces with the tools at hand, and you must explicitly call the *single abstract method*, as shown in Example 2-4.

Example 2-4. Lambdas in JavaScript versus Java

```
// JavaScript

var helloWorld = name => 'hello, ' + name + '!'
var result = helloWorld('Ben') ❶

// Java

Function<String, String> helloWorld = name -> "hello, " + name +
"!";
var result = helloWorld.apply("Ben"); ❷
```

❶ In JavaScript, functions are objects. But can call them directly by providing the arguments on parenthesis on the variable itself.

❷ In Java, a "function" is an object, too. But you need to call its *single abstract method* explicitly.

The call to the *single abstract method* might not be as concise as in other languages. But such verbosity allows for a backward-compatible way of calling lambdas that's familiar with Java developers, without the need to change the language itself.

Lambdas Versus Anonymous Classes

As a Java developer, you are most likely familiar with *anonymous inner classes*: the combined declaration and instantiation of types. An interface or extended class can be implemented "on-the-fly" without needing a separate Java class. On the surface, an anonymous class looks quite similar to lambda expressions, especially in Example 2-5.

Example 2-5. Anonymous class

```
// FUNCTIONAL INTERFACE (implicit)

interface HelloWorld {
  String sayHello(String name);
}


// AS ANONYMOUS CLASS

var helloWorld = new HelloWorld() {

  @Override
  public String sayHello(String name) {
    return "hello, " + name + "!";
  }
};

// AS LAMBDA

HelloWorld helloWorldLambda = name -> "hello, " + name + "!";
```

So are lambda expressions just *syntactic sugar* for anonymous classes for functional interfaces?

SYNTACTIC SUGAR

Syntactic sugar describes features that are additions to a language to make your life as a developer “sweeter.” Certain constructs can be expressed more concisely or clearly, or in an alternative manner.

Peter J. Landin coined the term in 1964[2], describing how the keyword `where` replaced λ in an ALGOL-like language.

Java’s `import` statement, for example, allows you to use types without their fully qualified names. Another example is type inference with `var` for references or the diamond operator <> for generic types. Both features simplify your code for “human consumption.” The compiler will “desugar” the code, though, it can deal with its “bitterness.”

Even though it might look like *just* syntactic sugar, it’s much more in reality. The *real* difference — besides verbosity — lies in the generated bytecode, as seen in Example 2-6, and how the runtime handles it.

Example 2-6. Bytecode differences between anonymous classes and lambdas

```
// ANONYMOUS CLASS

0: new #2 // class Anonymous$1 ❶
3: dup
4: invokespecial #9 // Method Anonymous$1."<init>":()V ❷
7: astore_1
8: return


// LAMBDA

0: invokedynamic #7, 0 // InvokeDynamic #0:sayHello:()LHelloWorld;
❸
5: astore_1
6: return
```

❶ A new object of the anonymous inner class `Anonymous$1` is created in the surrounding class `Anonymous`.

❷ The constructor of the anonymous class is called. Object creation is a two-step process in the JVM.

The `invokedynamic` opcode hides the whole logic behind creating

❸ The `invokedynamic` opcode hides the whole logic behind creating the lambda.

The anonymous version creates a new object of the anonymous class `Anonymous$1`, resulting in three opcodes:

`new`

Create a new uninitialized instance of a type.

`dup`

Put the value on top of the stack by duplicating it.

`invokespecial`

Call the constructor method of the newly created object to finalize intialization.

The total count ignores `astore_1`, because both versions store a reference into a local variable.

The lambda version only needs a single opcode: `invokedynamic`, delegating the actual creation to the JVM.

THE `INVOKEDYNAMIC` INSTRUCTION

Java 7 introduced this JVM opcode to allow dynamic method invocation methods. That allows the support of dynamic languages, like Groovy or JRuby. `invokedynamic` is a flexible invocation variant because its actual target is unknown on class-loading. Instead of linking dynamic methods — like lambdas — at compile-time, the JVM links a dynamic call site with the actual target method instead.

The runtime uses a so-called "bootstrap method" [3] (BSM) to link it and return a method handle on first-call. This way, the JVM can optimize lambda creation with different strategies, like dynamic proxies, anonymous inner classes, or MethodHandles. It's like using reflection in your code but safer and directly done by the JVM.

Another difference between lambdas and anonymous inner classes is their respective scope. An inner class creates a new scope, hiding its local variables from the enclosing one. Also, the keyword `this` references the instance of the inner class itself, not the surrounding scope. Lambdas, on the other hand, live fully in their surrounding scope. Variables can't be re-declared with the same name, and `this` refers to the instance the lambda was created in, if not `static`.

As you can see, lambda expressions are *not* syntactic sugar. That allows the JVM to optimize your functional code in new ways, even allowing the reuse of lambdas at the JVM's discretion.

Method References

Another syntax-change introduced with Java 8 is *method references*. It's shorthand syntactic sugar, using the new `::` (double-colon) operator, to reference an existing method in lieu of creating a lambda expression calling that method. If the input and output arguments match, *method references* allow you to streamline your functional code by eliminating the need for explicitly creating lambdas for dealing with already existing methods.

There are four types of method references, as listed in Table 2-1.

Table 2-1. The different types of method referer

e
n
c
e
s

Type	Syntax	Example
Static method	`ClassName::staticMethodName`	`List::of`
Instance of method (specific)	`variable::instanceMethodName`	customer::compareTo
Instance of method (arbitrary)	`ContainingType::methodName`	`String::toUpperCase`
Constructor	`ClassName::new`	`ArrayList::new`

Example 2-7 shows how a Stream pipeline's readability benefits from converting the lambdas to method references. Don't worry! You will learn about Streams later on in this book, just think of it as a fluent call with lambda accepting methods.

Example 2-7. Stream lambdas and method references

```
List<Customer> customers = ...;

// Lambdas

customer.stream()
        .filter(customer -> customer.isActive())
        .map(customer -> customer.getName())
        .map(name -> name.toUpperCase())
        .peek(name -> System.out.println(name))
        .toArray(count -> new String[count]);

// Method References
```

```
customer.stream()
        .filter(Customer::isActive)
        .map(Customer::getName)
        .map(String::toUpperCase)
        .peek(System.out::println)
        .toArray(String[]::new);
```

You can even use method references for constructs you don't usually use in *normal* Java, like casting objects, as seen in Example 2-8.

Example 2-8. Casting objects with a method reference

```
List<Object> maybeCustomers = ...;

maybeCustomer.stream()
        .filter(Customer.class::isInstance)
        .map(Customer.class::cast)
        ...
```

The `instanceof` operator and casting by using `(Customer)` can both be done by calling the appropriate methods on the `class` itself, allowing you to use both as method references.

Replacing lambdas with method references removes a lot of unnecessary noise without compromising the readability or understandability of your code. There is no need for the input arguments to have actual names and call their methods explicitly. The method references communicate the same amount of information to the reader in fewer chars typed and read. Also, modern IDEs usually provide you with a "quick fix" to convert lambdas to a method reference, if applicable.

(Almost) Pure Lambdas and Effectively final Variables

In Figure 1-1 you were introducted to the concept of *pure* — self-contained — functions that won't affect any outside state. Lambdas follow the same gist, but not only for conceptional or paradigmatic reasons. As mentioned before, the JVM tries its best to optimize lambdas with different strategies based on their actual usage pattern. But lambdas can "capture" values from their defining scope and have to be treated differently from *pure* lambas. That means their body can access variables from their creation scope, even if the lambda itself is no longer in that scope for these variables, as seen in Example 2-9.

Example 2-9. Lambda variable capture

```
void capture() {
  var theAnswer = 42; ❶
  Runnable printAnswer = () -> System.out.println("the answer is
" + theAnswer); ❷

  run(printAnswer);
}

void run(Runnable r) {
  r.run(); ❸
}
```

❶ The variable `theAnswer` is declared in the scope of `capture()`.

❷ The lambda `printAnswer` captures the variable in its body.

❸ The lambda can be run in another method and scope but still has access to `theAnswer`.

The big difference between *capture* and *non-capture* lambdas are the optimization strategies of the JVM. If no variables get captured, a lambda might end up being a simple `static` method behind the scenes, beating out the performance of alternative approaches like anonymous classes. The implications of capturing variables on performance are not as clear.

There are multiple ways the JVM might translate your code if it captures variables, leading to additional object allocation, affecting performance, and garbage-collector times. That doesn’t mean that capturing variables is inherently a bad design choice. If your requirements need the least amount of allocations or best performance possible, you should avoid unnecessary capturing, though. But try to refrain from premature optimizations just to avoid some overhead allocations. The main goal of a more functional approach should be improved productivity, more straightforward reasoning, and more concise code.

Effectively final

The JVM has to make special considerations to use them safely and achieve the best performance possible. That's why there's an important requirement regarding variables: only *effectively* `final` variables are allowed.

In simple terms, it represents an immutable reference that isn't allowed to change after its initialization. Any variable used by a lambda *must* be `final`, either by explicitly using the `final` keyword or by not changing after their initialization, making them *effectively* `final`. Be aware that this requirement is actually for the *reference* to a variable and *not* the underlying data structure itself. A reference to a `List<String>` might be `final`, but you can still add new items, as seen in Example 2-10.

Example 2-10. Change data behind a `final` variable

```
final List<String> list = new ArrayList<>();

list.add("adding is fine"); ❶

// WON'T COMPILE
list = List.of("assigning", "another", "List", "is", "not"); ❷
```

❶ The variable `list` is explicetly `final`, but new values can still be added.

❷ Reassigning the variable is prohibited and won't compile.

The simplest way to test whether a variable is *effectively* `final` or not is by making it explicitly `final`. If your code still compiles with the additional `final` keyword, it will compile without it. So why not make every variable `final`? Because it will add a lot of unnecessary noise and verbosity to your code. The compiler ensures that "out-of-body" references are *effectively* `final`, and the keyword won't help with actual immutability anyways.

WARNING

If you run any of the shown *effectively* `final`-related examples in `jshell`, they might not behave as expected. That's because `jshell` has special semantics regarding top-level expressions and declarations, which affects `final` or effectively `final` values at top-level[4]. Even though you can reassign any reference — making it non-effectively `final` — , you can still use them in lambdas, as long as you're not in the top-level scope.

In "The final keyword", you will learn more about the `final` keyword and its implications on your code and performance characteristics.

Re-finalizing a reference

Sometimes a reference might not be effectively `final`, but you still need them to be available in a lambda. If refactoring your code isn't an option, there's a simple trick to *re-finalize* them. Remember, the requirement is just for the reference and not the underlying data structure itself. So you can create a new *effectively* `final` reference to the non-*effectively* `final` variable, as shown in Example 2-11.

Example 2-11. Re-finalize a variable

```
var nonEffectivelyFinal = 1000L; ❶
nonEffectivelyFinal = 9000L; ❷

var finalAgain = nonEffectivelyFinal; ❸

Predicate<Long> isOver9000 = value -> value > finalAgain;
```

❶ At this point, `nonEffectivelyFinal` is still *effectively* `final`.

❷ Changing the variable after its initialization makes it unusable in lambda.

❸ By creating a new variable and not changing it after its initialization, you "re-finalized" the reference to the underlying data structure.

That's a neat trick that's good to know. But it's still only another "band-aid." And needing a band-aid means you scraped your knees first. So the best approach is trying not to need a band-aid at all. Refactoring or redesigning your code should always be the preferred option.

At first, the *effectively* `final` requirement might look like an additional burden. But it will force you to think in a more “pure” and side-effect-free way about lambdas. Instead of capturing “out-of-body” variables, your lambdas should be self-sufficient and require all necessary data as arguments. That automatically leads to more reasonable code, increased reusability, and allows for easier refactoring.

Batteries Included

Contrary to other functional languages, every lambda in Java must be backed by an actual interface. To not start your functional toolset by zero, the JDK provides over 40 different functional interfaces that can be grouped into four different categories:

- *Functions* accept arguments and return a result.
- *Consumers* only accept arguments and do *not* return a result.
- *Suppliers* do *not* accept arguments but return a result.
- *Predicates* accept arguments, test an expression, and return a *boolean* primitive.

The Big Four

The four different categories map directly to four functional interfaces (and their variants) present in the `java.util.function`:

Function<T, R>

One of the most central functional interfaces. It represents a “classical” function with a single input and output parameter, as shown in Figure 2-1. The input and output types can be identical, but in “Function Arity” you will learn about specialized functional interfaces with identical types.

Figure 2-1. Function<T, R>

`Consumer<T>`

As the name suggests, it *consumes* an input parameter, but doesn't return anything, as shown in Figure 2-2. Even though the sole consumption of a value in an expression might not fit into "pure" functional concepts, it's an essential component to elevate a more functional coding style in Java.

Figure 2-2. Consumer<T>

`Supplier<T>`

The antagonist to a `Consumer<T>`. It doesn't need any input parameter, but it returns a single value of type `T`, as shown in Figure 2-3.

Figure 2-3. Supplier<T>

`Predicate<T>`

A specialized function returning a `boolean` primitive, as shown in Figure 2-4. It's often used for decision-making, like `filter` methods of the functional pattern *map/filter/reduce* you will learn more about later on.

Figure 2-4. Predicate<T>

You can write a lot of functional code with just these four interfaces alone. In Example 2-12 you see all of them in action, combined in a Stream pipeline, which you will learn about in [Link to Come].

Example 2-12. The big four functional interfaces

```
Supplier<List<String>> lazy = ❶
  () -> List.of("apples",
                "oranges",
                "pear",
                "ananas",
                "banana");

lazy.get()
    .stream()
    .filter(in -> in.startsWith("a")) ❷
    .map(String::toUpperCase) ❸
    .forEach(System.out::println); ❹

// Output:
// APPLES
// ANANAS
```

❶ A `Supplier<List<String>>` providing a lazily created `List<String>`.

❷ A `Predicate<String>` filtering elements.

❸ A `Function<String, String>` returning the current element in uppercase.

❹ A `Consumer<String>` printing the current element.

Specialized Functional Interfaces

If there are just four categories, why does Java provide over 40 different functional interfaces? The answer lies in Java's type system. Functional interfaces are, well, interfaces, and lambda expressions are implementations of these interfaces. Type inference makes it easy to forget that you can't simply cast between interfaces, even if the method signatures are identical, like in Example 2-13.

Example 2-13. Casting between Functional Interfaces

```
Function<String, Long> fn = in -> 3L; ❶

interface CustomFunction { ❷
  Long apply(String value);
}

var customFn = (CustomFunction) fn; ❸
// => throws java.lang.ClassCastException
```

❶ A simple function you might want to cast later.

❷ A functional interface matching the function signature.

❸ Attempts to cast between the types will throw a `java.lang.ClassCastException`, regardless of a being a generic or non-generic implementation.

```
jshell> IntConsumer primitive = i -> System.out.println(i)
primitive ==> $Lambda$25/0x0000000800c0bc40@48533e64

jshell> Consumer<Integer> boxed = i -> System.out.println(i)
boxed ==> $Lambda$21/0x0000000800c0a800@34c45dca

jshell> IntConsumer fake = (IntConsumer) boxed
|  Exception java.lang.ClassCastException: class
REPL.$JShell$12$$Lambda$21/0x0000000800c0a800 cannot be cast to
class java.util.function.IntConsumer
(REPL.$JShell$12$$Lambda$21/0x0000000800c0a800 is in unnamed
module of loader
jdk.jshell.execution.DefaultLoaderDelegate$RemoteClassLoader
@2f0e140b; java.util.function.IntConsumer is in module java.base
of loader 'bootstrap')
|        at (#7:1)

jshell> IntConsumer fake = (IntConsumer) boxed::accept
fake ==> $Lambda$26/0x0000000800c0e410@484b61fc

jshell>
```

That’s why the JDK provides you with a lot of variations of functional interfaces, to give context-specific types, that express a certain intended use by their name alone and signature alone.

Function Arity

The concept of *arity* describes the number of operands taken by a function, regardless of the function being in logic, mathematics, or in our case, programming.

The number of operands in a Java method signature is fixed, so there must be an explicit functional interface for every arity. That's why the JDK provides additional interfaces for certain arities, including even more straightforward interfaces for identical input and output types, as listed in Table 2-2.

Table 2-2. Java Arity-based Functional Int

e
r
f
a
c
e
s

Arity	Specialized Type	Super Interface
1	`UnaryOperator<T>`	`Function<T, T>`
2	`BiFunction<T, U>`	-
2	`BinaryOperator<T>`	`BiFunction<T, T, T>`
2	`BiConsumer<T>`	-
2	`BiPredicate<T,U>`	-

You can create higher arities yourself, as you will see later on in Example 2-19.

These specialized interfaces will help you write more concise code, especially if they implement a common super interface. But be aware when you design APIs with these types. If your code requires an `UnaryOperator<String>`, it won't be compatible with `Function<String,String>`. The other way around works, though, as seen in Example 2-14.

Example 2-14. Java arity compatibility

```
UnaryOperator<String> unary = String::toUpperCase;

Function<String, String> func = String::toUpperCase;
```

```java
void acceptsUnary(UnaryOperator<String> unary) { ... };

void acceptsFunction(Function<String, String> func) { ... };

acceptsUnary(unary); // OK
acceptsUnary(func); // COMPILE-ERROR

acceptsFunction(func); // OK
acceptsFunction(unary); // OK
```

The increased verbosity of designing your methods to accept shared `super` interfaces in the signature is an acceptable trade-off, in my opinion, because it maximizes usability and doesn't restrict an argument to a specialized functional interface. But when creating a lambda though, the specialized type allows for more concise code without losing any expressiveness.

Primitive Types

So far, most functional interfaces have a generic type definition. But primitive types can't be used as generic types (yet). That's why there are specialized functional interfaces for primitives.

> **PROJECT VALHALLA AND SPECIALIZED GENERICS**
>
> The OpenJDK Project Valhalla is an experimental JDK project to develop multiple changes to the Java language itself. One of them that's quite relevant to simplifying lambdas is "specialized generics." Right now, generic type arguments are constrained to types that extend `java.lang.Object`, meaning that they are not compatible with primitives. Your only option is auto-boxed types like `java.lang.Integer`, etc., which has performance implications and other pitfalls compared to using primitives directly.
>
> The timeline of the project isn't clear yet. It was created in 2014, and in March 2020, the team behind it created five distinct prototypes to tackle the associated aspects of the problems.

You *could* use a generic functional interface for the object wrapper type and let auto-boxing take care of the rest. But auto-boxing isn't *free* and can have a performance impact. That’s why many of the functional interfaces provided by the JDK deal with primitive types to avoid auto-boxing. Such primitive functional interfaces aren’t available for all primitives, though. They are mostly concentrated around the numeric primitives `int`, `long`, and `double`. In Table 2-3 you see the available functional interfaces for `int`, but there are equivalent types available for `long` and `double`.

Table 2-3. Functional Interfaces for integer p

ri
m
it
iv
e

Functional Interface	**Boxed Alternative**
`IntFunction<R>`	`Function<Integer, R>`
`IntUnaryOperator`	`UnaryOperator<Integer>`
`IntBinaryOperator`	`BinaryOperator<Integer>`
`ToIntFunction<T>`	`Function<T, Integer>`
`ToIntBiFunction<T, U>`	`BiFunction<T, U, Integer>`
`IntConsumer`	`Consumer<Integer>`
`ObjIntConsumer<T>`	`BiConsumer<T, Integer>`
`IntSupplier`	`Supplier<Integer>`
`IntPredicate`	`Predicate<Integer>`
`IntToDoubleFunction`	`Function<Integer, Double>`
`IntToLongFunction`	`Function<Integer, Long>`

Only a single type is available for `boolean`: `BooleanSupplier`.

Functional interfaces for primitives aren't the only special consideration in the new functional parts of Java to accommodate primitives. As you will learn later in this book, Streams and Optionals provide specialized types, too, to reduce unnecessary overhead if needed.

Functional Programming Concepts in Java

"Functional Programming Concepts" introduced multiple concepts on a mostly theoretical level. Let's take another look at them regarding functions and lambdas as Java developers.

Pure Functions and Referential Transparency

The first concept is based on two guarantees that aren't necessarily bound to functional programming. It can easily be adopted in your imperative code without any functional programming.

Pure functions are:

- The *same* input will *always* create the same output. Therefore, repeated calls can be replaced by the initial result.
- They are self-contained without any kind of side effect.

Making your code predictable and reproducible brings in a multitude of advantages. Reasoning with your code becomes more straightforward, and it becomes unit-testable with ease. From a Java perspective, how can you achieve these beneficial properties?

Mutable state — if it crosses the boundaries of the functions — must be eliminated. Side-effects aren't restricted to mutable outside state, though. A simple `System.out.println(...)` call is a side-effect, even if it might look harmless. The reasoning behind this can be distilled to that repeated calls of the function with the same arguments can't be replaced with the result of the first evaluation. A good indicator for an *impure* method is a `void` return type. If a method doesn't return anything, all it does are side effects.

The last piece to the *pureness* puzzle is removing uncertainty. Like the example of calling `System.out.println(...)`, including any random code, like random number generators, the current date, etc., will result in unpredictable and unreproducible code, making it *impure*.

The same input will always generate the same output, which is called *referential transparency*. Hence, you can replace any subsequent calls with the same arguments with the previously calculated result.

This interchangeability allows for an optimization technique called *memoization*. Originating from the Latin word “memorandum" — *to be remembered* — , this technique describes “remembering” previously evaluated expressions. It trades memory *space* for saving computational *time*.

> **SPACE–TIME TRADE-OFF**
>
> Algorithms depend on two significant factors: *space* (e.g., memory) and *time* (e.g., computational or response time). Both might be available in vast quantities these days, but they are still finite. The *space-time trade-off* states that you can decrease one of the factors by increasing the other. If you want to save time, you need more memory by storing results. Or you can save memory by constantly recalculating them.

You’re most likely using the general concept already: *caching*. From dedicated cache-libraries, like Ehcache[5] to simple `HashMap`-based lookup tables, it’s all about “remembering” a value against a set of input arguments. But automatic memoization of functions or methods calls isn’t supported by the Java compiler. Some frameworks provide annotations, like `@Cacheable` in Spring[6] or `@Cached` in Apache Tapestry[7], and generate the required code automatically behind the scenes.

Building your own *memoization* by creating an “on-demand” lookup table requires the answer to two questions:

- How to identify the function input arguments uniquely?
- How to store the evaluated result?

If your function or method call has only a single argument with a constant `hashCode` or other deterministic value, you can create a simple `Map`-

based lookup table. But for multi-argument calls, you have to define how to create a lookup-key first.

Java 8 introduced multiple functional additions to the `Map<K, V>` type. The `computeIfAbsent(…)` is the perfect tool to implement memoization, as shown in Example 2-15.

Example 2-15. Memoization with `Map#computeIfAbsent`

```
ResultType expensiveCalculation(String arg0, int arg1) { ❶
    ...
}

private static final Map<String, ResultType> CACHE = new HashMap<>
(); ❷

ResultType memoized(String arg0, int arg1) { ❸
  var compoundKey = String.format("%s-%d", arg0, arg1); ❹
  return CACHE.computeIfAbsent(compoundKey,
                             key -> expensiveCalculation(arg0,
arg1)); ❺
}
```

❶ The time-consuming method for calculating a result.

❷ The results are cached in a simple `HashMap<String,ResultTyp>`. Depending on your requirements, there might be special considerations, like caching results per request in a web application or requiring a “time-to-live” concept. This example is supposed to show the simplest form of a lookup table.

❸ The “memoized” wrapper method around the cache. It has the same arguments as the calculation method because it delegates the work to it.

❹ The `ResultType` must be uniquely identifiable. Therefore, using a compound key in the case of multiple arguments makes sense.

❺ The mapping lambda is only evaluated if no value is associated with `compoundKey`.

The functional additions to `Map<K, V>` didn’t stop there. It provides the tools to create associations “on the fly.” But it also allows more fine-grained control if a value is already present. You will learn more about it in Chapter 4.

Immutability

Mutable state is the enemy of functional programming because most of its concepts rely on *immutable* data structures. Earlier in this chapter, in "(Almost) Pure Lambdas and Effectively final Variables" you've already learned about the restrictions Java lambdas impose on you to reduce side effects and enforce immutability on references, in the form of *effectively* `final` references.

Immutabily is a complex subject that you'll learn more about and its importance and how to utilize it properly — either with built-in tools or with a do-it-yourself-approach — in Chapter 3.

First-Class and Higher-Order

With Java *lambas* being concrete implementations of functional interfaces, they gain *first-class* citizenship — being usable as variable, arguments and return values — , as seen in Example 2-16.

Example 2-16. First-class Java Lambdas

```
UnaryOperator<Integer> quadraticFn = x -> x * x; ❶

quadraticFn.apply(5); ❷
// => 25


public UnaryOperator<Integer> multiplyWith(Integer multiplier) {
  return x -> multiplier * x; ❸
}

var multiplyWithFive = multiplyWith(5);

multiplyWithFive(5);
// => 25
```

❶ Assigning a Java lambda to the variable `quadraticFn`.

❷ It can be used like any other "normal" Java variable, calling the `apply` method of its interface.

❸ Returning a lambda is like returning any other Java variable.

Accepting lambdas as arguments and returning lambdas is essential for the next concept, *functional composition*.

Functional Composition

The idea of creating complex systems by composing smaller components is a cornerstone of programming. And functional composition is arguably one of the essential aspects of a functional programming mindset.

The general concept is simple: two functions are combined to build a new function. Smaller functions are composited into a larger chain of functions, creating a more complex system.

Functional interfaces can provide the necessary “glue methods” as `default` or `static` methods. In case of `Function<T, R>`, two `default` methods are available:

`<V> Function<V, R> compose(Function<? super V, ? extends T> before)`

Returns a composed function that first applies `before` to its input and then `this`.

`<V> Function<T, V> andThen(Function<? super R, ? extends V> after)`

The opposite of `compose(…)`, applying `this` first, and then `after`.

The direction of the composition is up to you. There’s no difference in the end result, as seen in Example 2-17.

Example 2-17. Functional composition direction

```
Function<String, String> removeLowerCaseA = in -> in.replace("a",
"");

Function<String, String> upperCase = String::toUpperCase;

var input = "abcd";

// Uppercase the String then remove the letter "a"
```

```
removeLowerCaseA.compose(upperCase)
                .apply(input);
// => "ABCD"

upperCase.andThen(removeLowerCaseA)
         .apply(input);
// => "ABCD"

// ---------------------------------------------

// Remove the letter "a" then uppercase the String

upperCase.compose(removeLowerCaseA)
         .apply(input);
// => "BCD"

removeLowerCaseA.andThen(upperCase)
                .apply(input);
// => "BCD"
```

Which direction to choose depends on the context and personal preference. I prefer `andThen(…)` because the resulting fluent method call-chain mirrors the logical flow of functions.

Composable Functional Interfaces in the JDK

Not every functional interface provides "glue methods" to allow composition, even if it would be sensible. And other provide methods that aren't just directly connecting the method's inputs and outputs but provide additional logic to simplify the method call-chain.

`Function<T, R>`

> `Function<T, R>`, and its specialized arities, like `UnaryOperator<T>`, support composition in both direction. The `Bi…` variants only support `andThen`.

`Predicate<T>`

> Predicates support various methods to compose a new Predicate with common operations associated with them: `and`, `or`, `negate`.

`Consumer<T>` and `Supplier<T>`

Both functional interfaces only support `andThen`.

Specialized primitive functional interfaces

The support for functional composition among the specialized functional interfaces for primitives is not on par with their generic brethren. And even among themselves, the support differs between the primitive types.

Currying

Currying — converting a function from taking multiple arguments into a sequence of functions that each take only a single argument — isn't natively supported by any functional interface. But you can create a helper to curry functions yourself, as shown in Example 2-18.

Example 2-18. Currying Helper

```
<T, U, R> Function<T, Function<U, R>> curry(BiFunction<T, U, R> fn)
{ ❶
    return t ->
              u -> fn.apply(t, u); ❷
}
```

❶ The curry method accepts a `BiFunction` and converts it to a `Function` returning another `Function`.

❷ The line break isn't necessary but better illustrates what's happening.

For easier use, such helper methods should be grouped together in a `static` class. If you need more arguments than two, you need to create the functional interface yourself. But you add the required `curry` method directly in the interface, as seen in Example 2-19 for ternary arity, instead of needing an extra helper class.

Example 2-19. Currying a TriFunction with a custom wrapper

```
@FunctionalInterface
public static interface TriFunction<A, B, C, R> {
```

```
  R apply(A a, B b, C c); ❶

  default Function<A, Function<B, Function<C, R>>> curry() { ❷
    return
      a ->
        b ->
          c -> apply(a, b, c);
  }
}


TriFunction<Double, Double, Boolean, Double> calculateFinalPrice =
❸
  (price, taxPercentage, includeTax) -> {
    if (includeTax == false) {
      return price;
    }

    return price + taxPercentage * price;
  };


Function<Double, Function<Double, Function<Boolean, Double> curried
=
  calculateFinalPrice.curry(); ❹

var finalPrice = curried.apply(100.0D) ❺
                        .apply(0.19D)
                        .apply(Boolean.TRUE);
// => 119.0
```

❶ The *single abstract method* is quite straightforward: accept three arguments and have a return value.

❷ Adding a `default curry()` method allows for easier currying.

❸ Creating a `TriFunction` as as simple as a `BiFunction`, nothing special about it.

❹ Calling `curry()` creates a nested `Function`, and you should use `var` instead of the explicit type.

❺ The curried variable now allows you to apply every single argument seperatly.

It might look quite unwieldy, but as many of the other concepts, they are interconnected with each other. In this case, *partial function application*.

Partial Function Application

The previous currying example can be extended to support the principle of partial application: applying only a subset of the required arguments.

You could use the `curry()` method of `TriFunction` as a starting point for partial application. But introducing additional `default` methods is a more flexible approach, as shown in Example 2-20.

Example 2-20. TriFunction Partial Application

```
@FunctionalInterface
public static interface TriFunction<A, B, C, R> {

  R apply(A a, B b, C c);

  default Function<A, Function<B, Function<C, R>>> curry() {
    ...
  }

  default BiFunction<B, C, R> partial(A a) { ❶
    return (b, c) -> apply(a, b, c);
  }

  default Function<C, R> partial(A a, B b) { ❶
      return c -> apply(a, b, c);
  }
}

TriFunction<Double, Double, Boolean, Double> calculateFinalPrice =
...

Function<Boolean, Double> basePrice =
calculateFinalPrice.partial(100.0D,

0.19D); ❷


var withTax = basePrice.apply(Boolean.TRUE); ❸

var withoutTax = basePrice.apply(Boolean.FALSE); ❸
```

❶ The `default partial(…)` methods create new lambas with less requirement arguments.

❷ The `TriFunction` can now be reduced to a *partially applied* version.

❸

The partially applied function is reusable and requires only a single argument.

Like other concepts, *parial application* is about the reusability of *pure* functions. It allows you to create a more generic pool of functionality that can be specialized as needed.

Takeaways

- Lambas are concrete implementations of *functional interfaces*.
- Their syntax is close to underlying mathematical notation. There are multiple verbosity levels possible, depending on the surrounding context and your requirements.
- Lambas are more than just *syntactic sugar*, with the JVM using the opcode `invokedynamic`.
- Method references are a concise alternative for matching method signatures and lambda definitions.
- Outside variables need to be *effectively* `final` to be used in lambdas, making the references immutable, but not the data structures themselves.
- The JDK provides a lot of different functional interfaces, including support for multiple functional techniques.
- Some edge cases are missing, but all tools to implement them yourself are provided.
- Primitives are supported by either using *auto-boxing*, or the specialized functional interfaces for `int`, `long`, `double`, and `boolean`.

1 The simplified version of `java.util.function.Predicate` is based on the source code for the latest Git tag of the LTS version at the time of writing: 17+35. You can check out the official source code repository to see the original file.

2 Landin, Peter J. (1964). "The mechanical evaluation of expressions." The Computer Journal. Computer Journal. 6 (4).

3 The class `java.lang.invoke.LambdaMetaFactory` is responsible for creating "bootstrap methods."

4 The official documentaton sheds some light on the special semantics and requirements for top-level expressions and declarations.

5 Ehcache is a widely-used Java cache library.

6 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html*

7 *https://tapestry.apache.org/5.7.2/apidocs/org/apache/tapestry5/annotations/Cached.html*

Chapter 3. Immutability

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

Immutability is a core tenet of functional programming and a prerequisite for many other concepts. It's a prevalent feature in many programming languages, like Haskell or Scala. Such languages make it a necessity or preferred way of handling data and not just an afterthought to their design. Like most other principles introduced in this book, immutability isn't restricted to functional programming and provides many benefits, regardless of your chosen paradigm.

This chapter will explain what it means for a data structure to be immutable and why it matters. You will learn about immutable types already available in the JDK and how to create your own immutable data structures, either with the tools provided by the JDK or with the help of third-party libraries.

Why is Immutability Important?

Immutability is the idea that data structures can no longer change after their creation. Many functional programming languages support it by design at their core. But the concept isn't bound to functional programming per se, and it has many advantages in any paradigm. The general concept provides elegant solutions to many problems, even outside of programming languages. For example, the distributed version control system *Git* essentially uses a tree of pointers to immutable blobs and diffs to provide a robust representation of historical changes.

Handling side effects is a really tough problem in programming — besides naming and cache invalidation[1]. Immutable data structures eliminate side effects because observed data structures won't and can't change, making their use less error-prone.

How to Change Data Structures That Can't Change

Instead of changing — or *mutating* — the original data structure, you have to create a new copy with the intended changes. Any existing reference to the original data structure will remain intact and contain the same unchanged data, no matter what. Change always has to be represented by a new data structure.

Even this simplified description of immutability already shows how it seems to be contrary to many *classical* and object-oriented Java patterns, especially *JavaBeans* and many forms of *Plain old Java objects (POJO)*. Your typical bean or plain object with its "getters" and "setters" encapsulates a mutable state and is therefore diametric to what immutability wants to achieve in the first place: *immutable state*.

JAVABEANS AND PLAIN OLD JAVA OBJECTS (POJOS)

A lot of confusion exists about JavaBeans and POJOs and their distinct properties. They are both ordinary Java objects, supposed to create reusability between components by encapsulating state. To make things even more complicated: every JavaBean is a POJO, but not every POJO is a JavaBean.

POJOs don't have any restrictions regarding their design. They just encapsulate the business logic state, and you can even design them to be immutable. It's up to you how you implement them and what matches your environment best. But they usually provide "getters" *and* "setters" for their fields to be more flexible in an object-oriented context with a mutable state.

JavaBeans, on the other hand, are a special kind of POJO. They have to oblige several conventions:

- `java.io.Serializable` must be implemented.
- Fields must be `private`.
- Fields are only accessible by getters and setters.
- A *no-arg* constructors must exist.

These requirements are necessary because JavaBeans were initially designed to be a standardized shareable machine-readable state between components. For example, every UI widget in your IDE could use a JavaBean to represent its state.

Justifying the Overhead of Immutability

Not being able to mutate data "in-place" can feel weird in Java at first. Compared to the more mutable nature of object-oriented code, why should you take the extra steps necessary to simply "change" a value? Such creation of new instances by copying data incurs a particular overhead that accumulates quickly for naive implementations of immutability.

Many functional programming languages and libraries mitigate this issue by reusing internal structures. Remember, if a data structure is genuinely immutable, its parts can't change either. So there's no reason why two different data structures can't share references to the same inner data. Such data structures are called "*persistent data structures*."

This optimization is especially beneficial if only a small change is required so that you can forgo an expensive deep copy. A naïve approach to changing immutable data structures will accumulate in terms of performance. In part 2 of this book, you will learn how to choose and use persistent data structures for everyday problems without sacrificing performance over usability.

Let's look at the advantages of immutability to understand better why immutable data structures worth the possible overhead and using a new approach to handle data structures.

Predictability

Data structures won't change without you noticing because they simply can't. As long as you reference a data structure, you know it is the same as at the time of its initialization. Even if you share that reference, no one can change your copy of it.

Validity

After initialization, a data structure is *complete*. It only needs to be verified once and stays valid (or invalid) indefinitely. If you need to build a data structure in multiple steps, the *builder-pattern*, shown later in "Step-by-Step Creation", decouples the building and initialization of a data structure.

No hidden side effects

Immutable data structures are always *as-is*. Even if it moves around a lot through different parts of your code or you send it to a third-party library out of your control, it won't change its values or surprise you with an unknown side effect.

Thread-safety

Without side effects, immutable data structures can move freely between thread boundaries. No thread can change them, so reasoning about your program becomes more straightforward due to no more unexpected changes or race conditions.

Cacheability and optimization

Because they are *as-is* right after creation, you can cache immutable data structures with ease of mind. Optimization techniques, like "Referential Transparency" and memoization are only possible with immutable data structure.

Change tracking

If every change results in a whole new data structure, you can track their history by storing the previous references. You no longer need to intricately track single property changes to support an *undo* feature. Restoring a previous state is as simple as using a prior reference to the data structure.

The State of Java Immutability

Java's initial design didn't include immutability as a deeply integrated language feature. Certain aspects of the language and its types are immutable, though. But compared to other languages, especially functional ones, it lacks some fundamentals. Still, all the pieces to create your own immutable types and data structures are available. And finally, Java 14 introduced a built-in language-level immutable data structure: *records*.

Let's take a look at the different immutable parts available in the JDK.

Immutable Types and Constructs of the JDK

Even if you might not know it yet, you’re already using immutable types in your Java programs. The reasons for their immutability might differ, like runtime optimizations or ensuring their correct usage. But regardless of their intentions, they’ll make your code safer and less error-prone.

java.lang.String

One of the first types every Java developer learns about is the `String` type. Strings are everywhere! That’s why it needs to be a highly optimized and safe type. And one of these optimizations is that it’s immutable.

The `String` type’s immutability creates an unavoidable overhead by requiring new objects for every change. That’s why Java developers learn early on not to overuse the + (plus) operator to concatenate `String` variables and literals. Every time you concatenate strings with +, a new `String` instance is created on the heap, needing memory, as depicted in Figure 3-1.

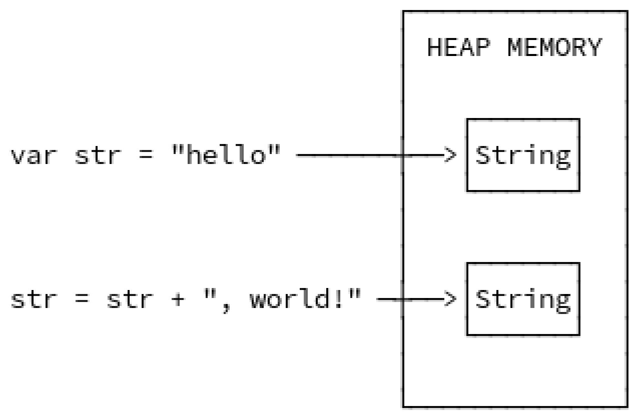

Figure 3-1. String memory allocation

The memory overhead of endless `String` creation can be a real burden on the runtime. That’s why the JVM uses multiple optimization techniques “behind the scenes” to reduce `String` creation, like replacing concatenations with a `java.lang.StringBuilder`, or even using the opcode `invokedynamic` to support multiple optimization strategies[2].

Being such a fundamental type, making it immutable is sensible for multiple reasons, even weighing in the possible downsides.

Concurrency

Having such a base type being *thread-safe-by-design* solves synchronization issues before they even exist. Concurrency is hard enough without worrying about a `String` to change without notice.

Caching

`String` literals get special treatment by the JVM. Thanks to *string pooling*, identical literals are reused and save precious heap space, as seen in Figure 3-2. If a `String` reference could change, it would change for everyone using a reference to it in the pool. But it's possible to allocate a new `String` by explicitly calling a constructor instead of creating a literal to circumvent pooling. There's usually no need for such an explicitly *new* `String`. It's supposed to highlight the benefits of `String` pooling thanks to immutability. The comparison of the object references confirms this immutability. But you should *never* compare `String` instances with == (double equal-sign), always use `equals` instead.

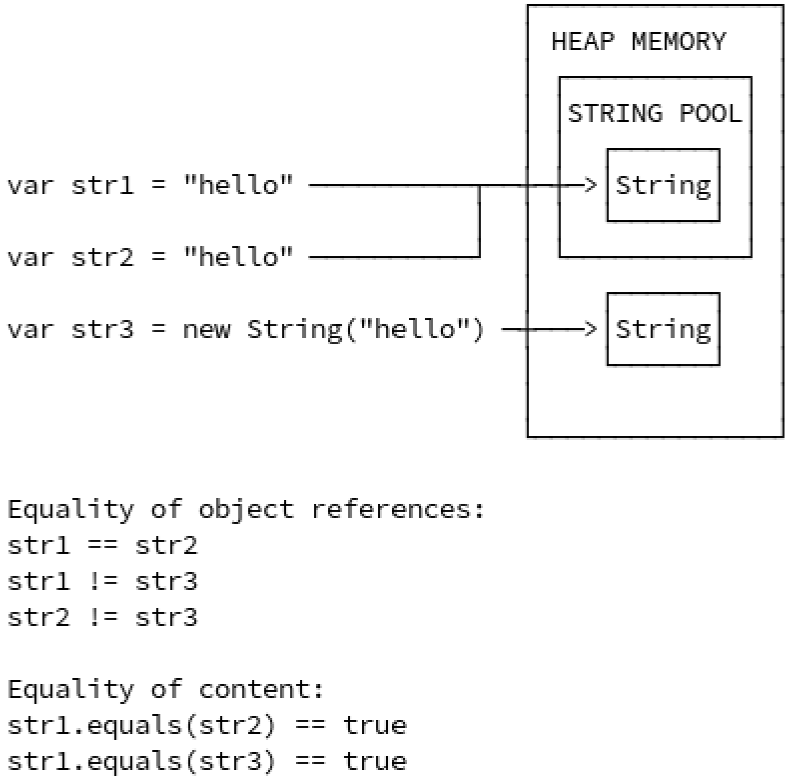

Figure 3-2. String pooling

Safety

: `String` objects are often used for arguments that aren't supposed to change between creation and actual use. Things like a username, connection strings, hash keys, or even class names in a `ClassLoader` are not supposed to change unexpectedly under any circumstance. Immutability removes the risk of race conditions, side effects, or even a simple unintended change.

However, the `String` type isn't "completely" immutable, at least from a strict point of view. It calculates its `hashCode` lazily due to performance considerations because it needs to read the whole `String` to calculate it. But it's still a *pure* function: the same `String` will result in the same `hashCode`. Such a *lazy* approach to achieve *logical immutability* to hide expensive *just in time* calculations require extra care during design and implementation to ensure the type is still thread-safe and predictable.

These properties make String something between a primitive and an object type, at least from a usability standpoint. Performance optimization possibilities and safety might have been the main reasons for its immutability. But the implicit advantages of immutability are a welcome addition to such a fundamental data structure.

Immutable Collections

Another fundamental and ubiquitous group of types that benefit significantly from immutability are collections. Java provides two types of immutable collections, or as it's called in the JDK, *unmodifiable* collections:

Unmodifiable wrapper views

Existing collections provide an unmodifiable view with a single call of one of the following generic static methods of the java.util.Collections type. Though, these unmodifiable wrappers provide only a shallowly immutable view of the provided collection. The listed signatures are shortened to improve readability:

- Collection<T> unmodifiableCollection(Collection<T> c)
- Set<T> unmodifiableSet(Set<T> s)
- List<T> unmodifiableList(List<T> list)
- Map<K,V> unmodifiableMap(Map<K,V> m)
- SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)
- SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,V> m)
- NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> s)

- `NavigableMap<K,V> unmodifiableNavigableMap(NavigableMap<K,V> m)`

SHALLOW IMMUTABILITY

The level of the immutability of data structures can vary greatly. That's why Java explicitly uses `Unmodifiable` as a type-name prefix, implying that no elements can be added or removed after creation. But the elements themselves aren't automatically immutable. To reach *deep* immutability, you need to use only completely immutable elements in such a collection.

Shallowly immutable collections provide only a subset of the guarantees that a deep immutable data structure could provide. They allow you to prepare a mutable collection as needed and return an unmodifiable view to anyone using it, preventing modification of the collection itself, but not its elements.

Unmodifiable Collections

Java 9 introduced `static` factory methods for unmodifiable collections on the types of `List`, `Set`, and `Map` itself: These factory methods don't rely on wrapping an existing collection. Instead, their elements are stored directly in an unmodifiable collection.

- `List<E> of(E… elements)`
- `Set<E> of(E… elements)`
- `Map<K, V> of(K k1, V v1, K k2, V v2, …)`

Immutable Math

Most simple math in Java can rely on *primitives*, like `int`, `long`, or even floating-point types like `float` or `double`. But the package `java.math` provides two immutable arbitrary-precision types for safer integer and decimal calculations: `java.math.BigInteger` and `java.math.BigDecimal`.

Just like with `String` before, why should you burden your code with the overhead of immutability? Because they allow side-effect-free calculations.

The immutable math types are still objects with the usual overhead and use more memory to achieve their precisions. But if calculation speed is not your limiting factor, you should always prefer the `BigDecimal` type for floating-point arithmetic due to its arbitrary precision[3].

The `BigInteger` type is the integer equivalent to `BigDecimal`, also with built-in immutability. Another advantage is the extended range of at least from $-2^{2,147,483,647}$ up to $2^{2,147,483,647}$ (both exclusive), depending on the actual implementation of the used JDK.

Java Time API (JSR-310)

Another set of types designed with immutability from the ground up is the Java Time API (JSR-310) introduced in Java 8. Before its release, you only had `java.util.Date`, `java.util.Calendar`, and `java.util.TimeZone` at your disposal for all your date- and time-related needs. Performing calculations were a chore and error-prone. That’s why Joda Time library became the de-facto standard for date and time classes before Java 8 and subsequently became the conceptional foundation for JSR-310.

Instead of three types, there now are multiple date- and time-related types with different precisions, with and without timezones, available. And all of them are immutable.

java.util.Locale

A `Locale` represents a specific geographical, political, or cultural region and its *locale-sensitive* rules, like number formatting conventions and date formats. It makes sense to make these rules unchangeable because they represent a specific ruleset shared across your application and shouldn’t change unexpectedly.

There are many pre-defined locales in the JDK, but you can also create your own.

Primitives & Primitive Wrappers

So far, you've learned mostly about immutable object types, but not everything in Java is an object. Java's *primitive* types — `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean` — are handled differently from object types. They are simple values that can only be initialized by either a literal or an expression. And by representing only a single value, they are practically immutable.

Besides the primitive types themselves, Java provides corresponding object wrapper types. They encapsulate their respective primitives in a concrete object type to make them usable in scenarios where primitives aren't allowed, like generics. Otherwise, *autoboxing* — the automatic conversion between the object wrapper types and their corresponding primitive type — could lead to inconsistent behavior.

Enums

Java enums are special types consisting of constants. And constants are, well, *constant*, and therefore immutable. Besides the constant values, an enum can contain additional fields which aren't implicitly constant.

Usually, `final` primitives or Strings are used for these fields, but no one stops you from using a mutable object type or a setter for a primitive. But it will most likely lead to problems, and I strongly advise against it and instead rely on the immutability of enums. It's also considered a *code smell* [4].

The final keyword

Java's `final` keyword provides a certain form of immutability. But it's not a magic keyword to make a data structure immutable, though. What exactly does it mean for a reference, method, or class to be `final`?

The `final` keyword is a relative of the `const` keyword of the programming language `C`. It has several implications if applied to classes, methods, fields, or references:

- `final` classes cannot be subclassed.
- `final` methods cannot be overridden.
- `final` fields must be assigned *exactly* once — either by the constructors or on declaration — and can never be reassigned.
- `final` variable references behave almost like a field by being assignable *exactly* once — at declaration. This affects only the reference itself, not the referenced variable content.

The final performance myth

The `final` keyword and its implications are often associated with improved performance. The reasoning behind this assumption is that the compiler will inline calls to `final` methods and references more safely and effectively. But as always, most things are more complicated than they look on the surface.

The compiler won't magically inline your methods and references just because they're marked `final`. At runtime, there's no guarantee that class `X` compiles against the same version of `final` class `Y`; it depends on which version is loaded by it. That means no cross-class inlining at compile-time, making the `final` keyword often unnecessary, at least for performance reasons.

Being no help at compile-time doesn't automatically mean that there's no inlining at runtime, though. The *just-in-time* compiler (JIT) has more information available than the compiler, like which version of a class is actually loaded. That allows it to safely inline methods, regardless of the `final` keyword. It becomes a mere hint to improve runtime optimizations. But even without it, your code will be optimized.

Does final provide Immutability?

If performance isn't the best rationale to use the `final` keyword, maybe immutability is? `final` grants a particular form of immutability, but what kind of immutability depends on the context: a `final` class is guaranteed to be immutable by prohibiting inheritance, a `final` method can't change even if its class is subclassed, and `final` references can't be reassigned. However, their immutability might not be what you expect. Because the reference *itself* becomes immutable and not the underlying data structure. That means you can't reassign the reference but still change the data structure, as shown in Example 3-1.

Example 3-1. `final` reference

```
final List<String> fruits = new ArrayList<>(); ❶

System.out.println(fruits.isEmpty());
// => true

fruits.add("Apple"); ❷

System.out.println(fruits.isEmpty());
// => false

fruits = List.of("Mango", "Melon"); ❸
// => WON'T COMPILE
```

❶ The `final` keyword only affects the reference `fruits`, not the actually referenced `ArrayList`.

❷ The `ArrayList` itself doesn't have any concept of immutability, so you can freely add new items to it, even if its reference is `final`.

❸ Re-assigning a `final` reference is prohibited.

Using `final` classes or methods can be quite inconvenient and might interfere with future extensibility. The keyword signals a particular set of guarantees and how to use the code and shouldn't be applied without a real need or intention behind it. It's best to document why a particular aspect is `final` to understand better why the benefits of `final` outweigh the inconvenience it may cause in the long run.

For variable references, it's a little bit more complicated. In "Effectively final" you've learned about *effectively* `final` references and their necessity for lambdas. But even then, the compiler can detect if a reference behaves like a `final` reference even without adding the keyword explicitly. And most problems created by the lack of immutability come from the underlying data structure itself and not reassigned references. To make sure a data structure won't change unexpectedly as long as it's in active use, you must choose an immutable data structure, regardless of the immutability of its references.

Records

Java 14 introduced a preview feature that's an immutable addition, sometimes replacement, for POJOs and beans: *records*. It took a few iterations for the feature to leave preview in Java 16. Records are not just another type or technique you can use; it's a whole new language feature that we'll look at in detail in the next section.

Immutable Records

A `record` is a new way to define *data-only aggregator types* in the vein of *nominal tuples*.

TUPLES

Mathematically speaking, a tuple is defined as a finite ordered sequence of elements. In terms of programming languages, a tuple is a data structure aggregating multiple values or objects.

There are two different kinds of tuples. *Structural* tuples rely only on the order of the contained elements and are therefore only accessible by their indices, as seen in the following Python code.

```
apple = ("apple", "green")
banana = ("banana", "yellow")
cherry = ("cherry", "red")

fruits = [apple, banana, cherry]

for fruit in fruits:
  print "The", fruit[0], "is", fruit[1]
```

Java Records, on the other hand, can be categorized into the *nominal* tuples. They use names instead of indices to aggregate and access their values, as seen in the following Swift code:

```
typealias Fruit = (name: String, color: String)
let fruits: [Fruit] = [
   (name: "apple", color: "green"),
   (name: "banana", color: "yellow"),
   (name: "cherry", color: "red")]

for fruit in fruits {
  println("The \(fruit.name) is \(fruit.color)")
}
```

Their data is shallowly immutable and transparently accessible. The typical boilerplate of data classes is significantly reduced by generating accessors and data-driven methods like `equals` and `hashCode`. Even though the final version of JEP 395 does explicitly state the "war on boilerplate" as a *non-goal*, it's a "happy coincidence" many developers will appreciate.

First, we will look at the “pre-`record`" state of data types to better grasp what records have to offer. We will create a simple *User* type and transition it from a `class` to a `record` type. There are some “missing features” to records, preventing them from being a “one-size-fits-all” solution, but you will learn how to mitigate them. Finally, alternative approaches for immutable data structures will be introduced.

Data Aggregator Types before Records

As an example, let’s develop a simple “user” type from a “classic” POJO class to an “immutable” POJO, and finally, a `record`. It will be as simple as it gets, not unnecessarily prolonging the code: a username, an activity state, and a “last login,” as seen in Example 3-2.

Example 3-2. Simple User POJO

```
public final class User {

  private String username;
  private boolean active;
  private Optional<LocalDateTime> lastLogin;

  public User() { } ❶

  public User(String username,
              boolean active,
              Optional<LocalDateTime> lastLogin) { ❶
    this.username = username;
    this.active = active;
    this.lastLogin = lastLogin;
  }

  public String getUsername() { ❷
    return this.username;
  }

  public void setUsername(String username) { ❸
    this.username = username;
  }

  public boolean isActive() { ❷
    return this.active;
  }
```

```java
  public void setActive(boolean active) { ❸
    this.active = active;
  }

  public Optional<LocalDateTime> getLastLogin() { ❷
    return this.lastLogin;
  }

  public void setLastLogin(Optional<LocalDateTime> lastLogin) { ❸
    this.lastLogin = lastLogin;
  }

  @Override
  public int hashCode() { ❹
    return Objects.hash(this.username,
                        this.active,
                        this.lastLogin);
  }

  @Override
  public boolean equals(Object obj) { ❺
    if (this == obj) {
      return true;
    }

    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }

    User other = (User) obj;
    return Objects.equals(this.username, other.username)
           && this.active == other.active
           && Objects.equals(this.lastLogin, other.lastLogin);
  }

  @Override
  public String toString() { ❺
    return new StringBuilder().append("User [username=")
                              .append(this.username)
                              .append(", active=")
                              .append(this.active)
                              .append(", lastLogin=")
                              .append(this.lastLogin)
                              .append("]")
                              .toString();
  }
}
```

Constructors aren't necessary but are added for convenience. If any

❶ Constructors aren't necessary but are added for convenience. If any constructor with arguments exists, an explicit "empty" constructor should be added, too.

❷ As with the constructor, you need explicit getters to access the fields.

❸ The first iteration of the `User` type still has setter methods to better reflect a "traditional" POJO.

❹ Both `hashCode()` and `equals(…)` require dedicated implementations that depend on the actual structure of the type. Any changes to the type require both methods to adapt.

❺ The `toString()` method is another convenience addition that isn't explicitly needed. Just like the previous methods, it has to be updated every time the type changes.

That's ~70 lines of code for just holding three data fields. No wonder one of the most common complaints about Java is its *verbosity*, and *too much ceremony* to do "standard" things!

Converting it to an immutable variant reduces the `User` type slightly because you no longer need the setter methods, as shown in Example 3-3.

Example 3-3. Simple immutable User type

```
public final class User {

  private final String username; ❶
  private final boolean active; ❶
  private final Optional<LocalDateTime> lastLogin; ❶

  public User(String username,
              boolean active,
              LocalDateTime lastLogin) { ❷
    this.username = username;
    this.active = active;
    this.lastLogin = Optional.ofNullable(lastLogin);
  }

  public String getUsername() { ❸
    return this.username;
  }

  public boolean isActive() { ❸
    return this.active;
  }
```

```java
  public Optional<LocalDateTime> getLastLogin() { ❸
    return this.lastLogin;
  }

  @Override
  public int hashCode() { ❹
    // UNCHANGED
    ...
  }

  @Override
  public boolean equals(Object obj) { ❹
    // UNCHANGED
    ...
  }

  @Override
  public String toString() { ❹
    // UNCHANGED
    ...
  }
}
```

❶ Without “setters”, the fields can be declared `final`.

❷ Only a full “pass-through” constructor is possible because the fields must be set on object creation.

❸ The “getters” remain unchanged from the mutable variant.

❹ The supporting methods are unchanged, too.

By making the type immutable yourself, only the code of the setters and the empty constructor could be removed. That’s still quite a lot of code for holding three fields with not much additional functionality. Of course, we could remove all of the “ceremony” and use a simple class with three `public final` fields and a constructor. Depending on your requirements, that might be “just enough.” But the additional functionality, like equality comparison, or a correct `hashCode()` so it can be used in a `Set` or `HashMap`, are desirable features. Although we’re looking for a more general solution to aggregate data in an immutable construct. So let’s take a look at the new data aggregator in town!

Records to the Rescue

Records are a new kind of class in the Java language. They are supposed to be plain and immutable data-aggregator types, allowing you to concentrate on the data modeling aspect for the state you want to represent instead of the "ceremony" of full-fledged data-holder types like "traditional" POJOs. The best approximation to records is "nominal tuples." They aggregate an ordered sequence of values and provide access via names instead of indices.

With records, you can eradicate all the boilerplate code for the `User` type, as seen in Example 3-4.

Example 3-4. Simple immutable User type as a record

```java
public record User(String username,
                   boolean active,
                   Optional<LocalDateTime> lastLogin) {
}
```

That's it!

Record declarations consist of a header and a body:

```
// HEADER
[visibility] record [Name][<optional generic types>]
([components]) {
  // BODY
}
```

The header is similar to a `class` or `interface` header and consists of multiple parts:

Visibility

The header starts like a `class` definition, with an optional visibility keyword (`public`, `private`, `protected`).

`record` keyword

The keyword distinguishes the header from other type delcarations (`class`, `enum`, `interface`).

Generic types

Like classes or interfaces, records also support generic types.

Name

Record names follow the same rules as any other identifier, as defined in the *Java Language Specification* [5].

Components

The name is followed by a pair of parentheses containing the components of the record. Each one translates into a `private final` field and a `public` accessor. The components also represent the constructor of the record.

Functionality-wise, both immutable implementations are the same, thanks to the implicit functionality of *records*. The effectively single line of code will be translated by the compiler to a class similar to Example 3-3, but it extends `java.lang.Record` explicetly rather than `java.lang.Object` implicetly. The command `javap User.class` shows what the compiler does "behind-the-scenes," its output of the POJO, and the record is shown in Example 3-5, with reordered entries for easier comparison.

Example 3-5. Dissambled User.class POJO versus record

```
// IMMUTABLE POJO

public final class User {
  public User(java.lang.String, boolean, java.time.LocalDateTime);
  public java.lang.String getUsername();
  public boolean isActive();
  public java.util.Optional<java.time.LocalDateTime>
getLastLogin();
  public int hashCode();
  public boolean equals(java.lang.Object);
  public java.lang.String toString();
}
```

```java
// RECORD

public final class User extends java.lang.Record {
  public User(long, java.lang.String, boolean,
java.util.Optional<java.time.LocalDateTime>);
  public long id();
  public java.lang.String username();
  public boolean active();
  public java.util.Optional<java.time.LocalDateTime> lastLogin();
  public final int hashCode();
  public final boolean equals(java.lang.Object);
  public final java.lang.String toString();
}
```

One noticeable difference between the two versions is the constructor definition. A record defines its components with its constructor, so you can't use different types, like `Optional` fields combined with non-optional constructor arguments. But allowing such discrepancy between the constructor and the actual components would reduce the transparency of records. Having stricter but more transparent types will communicate their intent more clearly, and provide a more concise code.

Automatic Record Features

Records are as transparent as possible. That implies certain guaranteed properties and "automagically" provided functionality without writing trivial implementations repeatedly.

Getters, no setters

> All fields are private, but the record provides a `public` accessor with the same name and return type as the field. The standard POJO/bean prefix "`get`" is omitted.

Canonical constructor

> A constructor with the same signature as the header components is implicitly available.

Object identity

You don't have to implement `hashCode` or `equals` yourself. Records provide the "typical" implementations, meaning they are equal if their components are equal.

Object description

A "typical" `toString` implementation with all component values and names is available.

Generics and annotations

Like any other type, records support generics and annotations.

Most of the usual ceremony and verbosity are avoided by records "automagically" provided functionality. But you can still override any of them if your requirements differ from the default implementations. Even though that's a lot of functionality without any additional code, records are still lacking in some areas.

Custom and Compact Canonical Constructor

The automatically provided canonical record constructor assigns its components "as-is." But you can define your own canonical constructor to validate components, as seen in Example 3-6.

Example 3-6. Override canonical record constructor

```
public record User(long id,
                   String username,
                   boolean active,
                   Optional<LocalDateTime> lastLogin) {

  public User(long id,
              String username,
              boolean active,
              Optional<LocalDateTime> lastLogin) {

    if (id < 0) {
      throw new IllegalArgumentException("id must be positive");
    }

    Objects.requireNonNull(userName);
```

```
    Objects.requireNonNull(lastLogin);

    this.id = id;
    this.username = username;
    this.active = active;
    this.lastLogin = lastLogin;
  }
}
```

There's also a compact form that doesn't require you to provide the components again in its signature, and you won't need to assign the fields yourself, as shown in Example 3-7.

Example 3-7. Compact canonical constructor override

```
public record User(long id,
                   String username,
                   boolean active,
                   Optional<LocalDateTime> lastLogin) {

  public User { ❶

    if (id < 0) {
      throw new IllegalArgumentException("id must be positive");
    }

    Objects.requireNonNull(userName);
    Objects.requireNonNull(lastLogin);

    ❷
  }
}
```

❶ The constructor omits all arguments, including the parenthesis. That might look unusual, but it distinguishes the compact canonical constructor from an additional constructor without any arguments.

❷ The field assignments aren't allowed in the compact canonical constructor and will be assigned implicitly.

The compact canonical constructor is another record feature highlighting its intent to be as transparent and state-oriented as possible.

Missing Features

Records are precisely what they are supposed to be: *plain, transparent, shallowly immutable data-aggregators*. But compared to other available solutions, like beans or POJOs, they lack some features you might be used to:

- Additional State
- Inheritance
- Default values
- Step-by-step creation

Additional State

Allowing any additional opaque state is an obvious omission from records. They are supposed to be *data-aggregators* representing a transparent state. That's why any field added to the record body will result in a compiler error. If you require additional state, records might not be the solution you're looking for. Methods can be added, as they don't introduce additional state and can only work with the components already present in the record.

Inheritance

Records are `final` types that already extend `java.lang.Record` behind-the-scenes, as previously seen in Example 3-5. With Java not allowing multi-inheritance, they can't extend any other type. But that doesn't mean they can't implement any interfaces, though. That allows you to define record templates and share common functionality with `default` methods, as shown in Example 3-8.

Example 3-8. Using interfaces with records

```
interface Origin {

  float x(); ❶

  float y(); ❶
```

```
  default String origin() { ❷
    return String.format("(%.2f/%.2f)", x(), y());
  }
}


interface Area {

  float area(); ❸
}


record Point(float x,
             float y) implements Origin { }


record Rectangle(float x,
                 float y,
                 float width,
                 float height) implements Origin, Area {

  float area() { ❸
    return width() * height();
  }
}


record Circle(float x,
              float y,
              float radius) implements Origin, Area {

  float area() { ❸
    return (float) MATH.PI * radius() * radius();
  }
}
```

❶ The interface defines the components of an implementing record as simple methods with the correct names

❷ Shared functionality is added with `default` methods.

❸ Method signatures in interfaces must not interfere with any implementing record type.

Interfaces can provide a few left-out pieces of the missing inheritance, and it might be tempting to create intricate hierarchies and interdependencies between records. But structuring your record types this way will create cohesion between them that's not in the original spirit of records to be simple data aggregators defined by their state.

Component Default Values

Java, unlike many other languages, doesn't support default values for any constructor or method arguments. A record provides only its canonical constructor with all components automatically, so you must add any additional constructors with default values yourself, as seen in Example 3-9.

Example 3-9. Multiple record constructors

```
public record User(long id,
                   String username,
                   boolean active,
                   Optional<LocalDateTime> lastLogin) {

  public User(long id, String username) {
    this(id, username, false, Optional.empty());
  }
}
```

Step-by-Step Creation

One of the advantages of immutable data structures is the lack of "half-initialized" objects. But not every data structure is initializable at once. Instead of reaching around a mutable data structure, you can use the *builder pattern*. Even though it was incepted as a solution to recurring object creation problems in object-oriented programming, it's highly beneficial for creating immutable data structures.

> **TIP**
>
> The *builder design pattern* is one of the design patterns introduced in the book *Design Patterns: Elements of Reusable Object-Oriented Software* [6] by the "Gang of Four," referring to Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Their book is the de-facto reference book on object-oriented design patterns, including 23 different patterns.

By separating the construction of the data structure from its representation, the data structure itself can be a simple as possible, making the pattern an excellent match for records. Any required logic, or validation, can be encapsulated into a (multistep-)builder. The previously used `User` record can be complemented by a simple builder, as shown in Example 3-10.

Example 3-10. User Builder

```
public final class UserBuilder {

  private final long   id; ❶
  private final String username; ❶

  private boolean       active; ❷
  private LocalDateTime lastLogin; ❷

  public UserBuilder(long id, String username) {
    this.id = id;
    this.username = username;
    this.active = true; ❸
  }

  public UserBuilder active(boolean isActive) { ❷
    if (this.active == false) { ❹
      throw new IllegalArgumentException("...");
    }

    this.active = isActive;

    return this; ❺
  }

  public UserBuilder lastLogin(LocalDateTime lastLogin) { ❻
    this.lastLogin = lastLogin;

    return this;
  }

  public User build() { ❼
    return new User(this.id,
                    this.username,
                    this.active,
                    Optional.ofNullable(this.lastLogin));
  }
}
```

```
var builder = new UserBuilder(42L,
"ben").lastLogin(LocalDateTime.now());
                                                 .active(false);  ❽

var user = builder.build(); ❾
```

❶ Required fields should be `final` and required in the constructor.

❷ Field that can be changed during building need setter-like methods.

❸ Explicit default values are possible, reducing the required code for creation.

❹ Validation logic is bound to the specific setter-like method and not accumulated in any constructor.

❺ Returning `this` creates a fluent API for the builder.

❻ `Optional` fields can use their explicit types, and only change into an `Optional` during `build()`.

❼ If you're done building, calling `build()` will create the actual immutable `User` record. Usually, the builder should validate its state if necessary.

❽ The build process is fluent, and you can reach the builder around like any other variable.

❾ Finally, create the immutable object by calling `build()`.

It's sensible to increase the adhesion between the type and its builder by placing the builder class directly in the corresponding type as a `static` nested class, as seen in Example 3-11.

Example 3-11. Builder as nested class

```
public record User(long id,
                   String username,
                   boolean active,
                   Optional<LocalDateTime> lastLogin) {

  public static final class Builder {
    // ...
  }
}

var builder = new User.Builder(42L, "ben");
```

It might seem non-sensical to use a record to achieve simplicity and immutability but still introduce the complexity of a builder. Why not use a full-fledged bean instead? Because even with the complexity of the builder, the concerns of creating the data, and using it, are separated. The record type is still usable without the builder. But the builder provides a flexible way to create it, if needed.

The builder pattern also allows for easier change management by introducing a copy-constructor to it. With it, you can easily initialize a builder with an existing record, make the appropriate changes, and create a new record, as seen in Example 3-12.

Example 3-12. Builder with copy constructor

```
public record User(long id,
                   String username,
                   boolean active,
                   Optional<LocalDateTime> lastLogin) {

  public static final class Builder {

    // ...

    public UserBuilder(User user) {
      this.id = user.id();
      this.username = user.username();
      this.active = user.active();
    }

    // ...
  }
}

var user = new User.Builder(42L, "ben");

var inactive = new User.Builder(user)
                       .active(false)
                       .build();
```

Another option to create a new record with a single change are `with`-methods. They take a single argument and create a new record base on `this` and the argument, as shown in Example 3-13.

Example 3-13. Record with-methods

```
public record User(long id,
                   String username,
```

```java
                     boolean active,
                     Optional<LocalDateTime> lastLogin) {

  public User withUsername(String newUsername) {
    return User(id(),
                newUsername,
                active(),
                lastLogin());
  }
}
```

Such "`with`" methods are a convenient way to realize a single component change. But if you want to change more than one component, using a fluent call with multiple `with` methods will create an intermediate instance on each call. In this case, a builder with a copy-constructor, or `with` methods with multiple arguments, is preferable due to less overhead required.

In part 2 of this book, I will introduce another option: reflection. You will learn how to build a generic tool for adapting records by their component names.

Record Serialization

Records, like classes, aren't serializable out-of-the-box, and you have to implement the `java.io.Serializable` interface. But unlike classes, records provide an optimized serialization strategy without any additional code.

Serialization of classes relies heavily on costly reflection to access the private state of a deserialized object. The process is customizable by implementing the methods `readObject` and `writeObject`, which are hard to get right and have led to many exploits in the past[7].

Records are only defined by their immutable state, allowing them to be reconstructable with their canonical constructor instead of reflection. Customizing that process isn't possible, leading to a safer serialization process, with the record taking back control of its serialized representation. The lack of reflection allows for better performance, too. Once the JVM derives the serialized form of a record, a matching instantiator can be cached.

> **NOTE**
>
> The serialization process consists of *serialization* (converting an object to a byte stream) and *deserialization* (reading an object from a byte stream). If not explicitly mentioned in the book, *serialiaztion* describes the whole process, not only one of its aspects.

Another interesting aspect of record serialization is the improved backward compatibility in case the type itself evolves. If the record gains a new component, it still deserializes from a previously serialized record. Any unknown component will pass its *default* value to the canonical constructor during deserialization. For example, a two-dimensional `record Point(int x, int y)` can evolve into the third dimension by adding `int z`, initialized as its default value, `0`. You can't run the corresponding code in JShell, because the internal class names won't be identical after replacing the record definition, so you have to run it with two different Java files, as seen in Example 3-14.

Example 3-14. Record evolution and serialization

```
// Serialize.java

import java.io.*;

public class Serialize {

  public record Point(int x, int y) implements Serializable { }

    public static void main(String... args) throws IOException {

      var point = new Point(23, 42);
      // => Point[x=23, y=42]

      var out = new ObjectOutputStream(new
FileOutputStream("point.data"));
      out.writeObject(new Point(23, 42));
      out.close();
    }
```

```
}


// Deserializa.java

import java.io.*;

public class Deserialize {

  public record Point(int x, int y, int z) implements Serializable
{ }

  public static void main(String... args) throws IOException,
ClassNotFoundException {

    var in = new ObjectInputStream(new
FileInputStream("point.data"));
    var point = in.readObject();
    // => Point[x=23, y=42, z=0]
  }
}
```

As mentioned in the introduction of "Immutable Records", records are in the vein of nominal tuples. That means they're based on their components' names and types, not their exact order. That allows you to change the order of the components without breaking its deserialization capabilities. Besides adding new components, you can also remove any of them.

Records appear to be solely defined by their components. From a non-serialization point of view, that's a valid argument. But the types aren't interchangeable on serialization! Even if two records are defined with identical components, you will encounter a `ClassCastException` if you try to deserialize into another type.

Final Thoughts on Records

Records derive their implementation for creating, accessing, comparing, and representing data from the description of its state in the form of its components. Even though that provides you with a great deal of simplicity, it's contrary to a fundamental concept of object orientation: *encapsulation*. It's an essential concept to manage and not expose complexity. But in the case of records, the abstractions are supposed to be simple. There aren't any "one-size-fits-all" solutions for data storage and state, and records aren't supposed to replace POJOs or other pre-existing data-aggregator types completely.

The imposed restrictions of records might seem arbitrary and even confining at first, and you might wish for more flexibility at some point. But the feature-set of records was chosen deliberately to have a new type of state representation, and *only* state. They are defined by their immutable components and should only rely on them. The simplicity of defining a new record prevents us from reusing an abstraction just because it might be more convenient than creating a new one. Implementing the "usual" `Object` methods (`hashCode`, `equals`, `toString`) is a chore, too, and can easily lead to bugs and has to be updated if the type changes.

Records might not be as flexible as POJOs or custom types. But flexibility usually means more complexity, and that leads to an increased bug surface. The best way to deal with it is to reduce the surface as much as possible. And records provide a lot of functionality "for free" and won't break if the type changes, reducing such surface immensely.

Third-Party Immutabilty

Even though you can save much boilerplate code with records, the need for an extra builder or copy-constructors is reintroducing repetitive boilerplate code. And every line of additional code can introduce bugs. That's why using a third-party library to create such boilerplate for you behind-the-scenes can be a relief, saving you from typing all the code, including convenience methods and validation.

There are two frameworks I want to mention that reduce the amount of code necessary for quite intricate immutable data structures: *Project Lombok* and *Immutables*.

Project Lombok

Project Lombok is a compehensive tool to reduce repetitive boilerplate code, like getters/setters, `null` checks, `equals/hashCode`, or `toString`. And luckily for us, immutables, too.

The `User` record can be defined as a Lombok `@Value` class, as seen in Example 3-15.

Example 3-15. User as Lombok @Value class

```
@Value
@Builder
public class User {
  @NonNull String              username;
  boolean                      active;
  Optional<LocalDateTime> lastLogin;
}
```

The annotation `@Value` is equivalent to adding multiple other Lombok annotations:

`@Getter`

Generate getters for all fields.

`@FieldDefaults(makeFinal=true, level=AccessLevel.PRIVATE)`

Fields will be `final` and `private`

`@AllArgsConstructor`

A constructor with all fields as arguments will be added, including `@NonNull` checks.

`@EqualsAndHashCode`

Default implementation for `equals` and `hashCode`.

`@ToString`

Default implementation for `toString`.

The `@Builder` annotation adds a builder based on the fields to the type. It will include `null`-checks for all `@NonNull` fields.

Immutables

The Immutables project utilizes annotation processors to generate immutable types based on your abstract types, either an abstract class or an interface, and their annotations, as shown in Example 3-16.

Example 3-16. User type as Immutable

```
@Value.Immutable
public interface User {

  long getId();

  String getPassword();

  @Value.Default
  default boolean isActive() {
    return false;
  }

  Optional<LocalDateTime> getLastLogin();
}
```

The annotation processor generates an implementation behind the scenes, named `ImmutableUser`, including everything you would expect, like serialization, builder classes, validation, default values, convenience methods for copying, style customization, JSON support, and much more.

Takeaways

- Immutability is a simple concept but requires a new approach to handling data and change.
- Lots of JDK types are already designed with immutability in mind.
- Records provide a new and concise way to reduce boilerplate for creating immutable data structures but deliberately lack certain

flexibility to be as transparent and straightforward as possible.

- Records aren't a "one-size-fits-all" solution. But you can easily create a builder or helper method to increase their flexibility.
- Third-party libraries also provide simple solutions to the missing pieces to get the most out of immutability (and more).

---

1 Phil Karton, an accomplished software engineer who for many years as a principal developer at Xerox PARC, Digital, Silicon Graphics, and Netscape, coined the quote, "There are only two hard things in Computer Science: cache invalidation and naming things." It became a mainstream joke in the software community over the years and is often amended by adding "one-off errors" without changing the count of two.

2 The JDK Enhancement Proposal (JEP) 280, "Indify String Concatenation", describes the reasoning behind using `invokedynamic` in more detail.

3 Arbitrary-precision arithmetic — also known as bignum arithmetic, multiple-precision arithmetic, or sometimes infinite-precision arithmetic — performs calculations on numbers whose digits of precision are only limited by the available memory, not a fixed number.

4 A *code smell* is a known code characteristic that might indicate a deeper problem. It's not a bug or error *per se*, but it might cause trouble in the long run. These *smells* are subjective and vary by programming language, developer, and paradigms. SonarSource, the well-known company that develops open-source software for continuous code quality and security, lists mutable enums as rule RSPEC-3066

5 See the Java Language Specification chapter 3.8 for the definition of valid Java identifier.

6 Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison Wesley.

7 The method `readObject` can execute arbitrary code instead of simply reading the object. Some related CVEs: CVE-2019-6503, CVE-2019-12630, CVE-2018-1851.

Chapter 4. Lazy Evaluation

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

In people, *laziness* is often seen as a character flaw. Some programming languages favor *laziness*, but does that mean they do less in a negative way, too?

In programming languages, *laziness* is the antagonist to *strictness* — or *eagerness* — of code evaluation. This chapter will show you how being *lazy* can improve performance. You will learn about the difference between *strict* and *lazy* evaluation and its impact on your code's design.

Laziness Versus Strictness

The *strictness* of a language describes the semantics of how expressions are *reduced* according to *lambda calculus*, as you've seen in "Lambda Abstractions". For example, *Haskell* is a functional programming language with *non-strict* semantics by default. Such semantics allow expressions to have a value even if one or more subexpressions fail to evaluate. This way, you can create control strucutues or infinite data sequences due to the separation of the *creation* and *consumption* of expressions.

Java is *strictly* evaluated like most modern multi-paradigm languages. *Strict* evaluation is *always* evaluating as soon as possible, like declaring or setting a variable, or passing an expression as an argument, as seen in Example 4-1.

Example 4-1. Strict Evaluation

```
var sum = 1 + 2;

aMethodCall(sum);

anotherMethodCall(1 + 2)
```

The variable `sum` evaluates as soon as it's defined, and `aMethodCall` receives the value `3` instead of the expression itself. Method arguments are *passed-by-value*, which means they're evaluated first before being passed to a method. That's why the expression `1 + 2` is evaluated before calling `anotherMethod`.

> **NOTE**
>
> Java's method arguments are always *pass-by-value*. In the case of non-primitive types, arguments are passed as *object-handles*, so-called *references*. But these are technically still *passed-by-value*, though, making the general terminology and semantics quite confusing.

Conversely, *lazy* evaluation is defined as not evaluating expressions until their values are needed. That means the declaration of an expression doesn't trigger its immediate evaluation, which makes Java lambda expressions the perfect match for *lazy* evaluation, as seen in Example 4-2.

Example 4-2. Lazy Evaluation

```
IntSupplier lazySum = () -> { 1 + 2 };

aMethodCall(lazySum);

anotherMethodCall(() -> { 1 + 2 })
```

The declaration of `lazySum`, or its inline equivalent, is a *strict* statement. But the expression itself isn't evaluated until explicitly called.

Strictness is about *doing things*, but *laziness* is about *considering things to do*.

How Strict Is Java?

Java is considered a strict language; everything evaluates immediately, like expressions and method arguments. But most programming languages are neither fully *lazy* nor *strict*, neither is Java. There are several noteworthy exceptions to the rule, both on language level and in the available types of the JDK.

Short-Circuit Evaluation

Language-integrated laziness is available in form of the logical *short-circuit* operators `&&` (double-ampersand / logcial *and*) and `||` (double-pipe / logical *or*). These operators evaluate their operands left-to-right only as required. If the logical expression is satisfied, the other operand isn't evaluated at all, as seen in Table 4-1.

Table 4-1. Evaluation of logical short-cir *cuit operators*

Operations	Value of **`leftExpr`**	Is **`rightExpr`** evaluated?
`leftExpr && rightExpr`	`true`	yes
	`false`	no
`leftExpr || rightExpr`	`true`	no
	`false`	yes

Despite functioning similarly to a control structure, the logical operands can't exist in a vacuum. They must always be part of another statement, like an `if`-block or a variable assignment, as seen in Example 4-3.

Example 4-3. Usage of logical short-circuit operators

```
// WON'T COMPILE
left() || right();


// COMPILES

if (left() || right()) {
    // ...
```

```
}

var result = left() || right();
```

Omitting the right-side operand evaluation is extremely helpful if the expression is costly or does have any side effects. Short-circuit evaluation is also a more convenient way to assign (effectively) `final` [1]. The alternative, splitting the declaration of the variable and assigning it in an `if-else` block, requires you to re-finalize the variable, as discussed in [Link to Come].

Control Structures

Control structures are responsible for changing the path taken through the instructions of your code. An `if-else` construct is a conditional branch, and strictly evaluating both parts indiscriminately would defeat its purpose. That lazy exception to the eager rules applies to all branching and loop structures:

Branching control structures

- `if-else`
- `? :` (ternary operator)
- `switch`
- `catch`

Loop strucutres

- `for`
- `while`

An absolutely strict language with non-lazy control structures is hard to imagine, if not impossible.

Lazy Types in the JDK

So far, the JDK's laziness was built into the language itself. But it also provides multiple types with a certain degree of laziness.

Lazy Maps

A common task for maps is checking if a key already has a value, and providing one if it's missing, as seen in Example 4-4.

Example 4-4. Checking a Map for a value

```
Map<String, User> users = ...;

var email = "john@doe.com";

var user = users.get(email);
if (user == null) {
  user = loadUser(email);
  users.put(email, user);
}
```

The code might vary depending on the actual `Map` implementation used, but the gist behind it is clear. Even though the code delay loading a user to the moment it's necessary, the `Map` type provides a more concise and functional alternative with `computeIfAbsent`.

Example 4-5. Compute an absent Map value

```
Map<String, User> users = ...;

var email = "john@doe.com";

var user = users.computeIfAbsent(email,
                                 this::loadUser);
```

It requires the desired key as its first argument and a mapper `Function<K, V>` as its second argument. The general theme of a functional approach is clearly visible: declarative code highlighting *what* is happening, without the verbosity of *how* it's actually done.

Streams

Java Streams are the perfect example of lazy functional pipelines. You can define an intricate Stream with expensive operations that will only evaluate if absolutely necessary. This laziness reduces the required work by separating the definition of an expression and its actual evaluation in a data processing pipeline. Streams and their lazy approach to data processing will be explained in detail in the next chapter, [Link to Come].

Optionals

Optionals are a *non-lazy* way of handling `null` values. Their approach is similar to Streams, but they evaluate strictly compared to Streams. They allow a lazy alternative value if necessary, though, in the form of the `Optional#orElseGet(Supplier<? extends T> supplier)` method. See Chapter 5 for a detailed introduction of Optionals and more information on how to use them.

Lambdas and Higher-Order Functions

Lambdas are a great way to introduce *laziness* on a code level. Their declaration is a statement, and therefore *strictly* evaluated. But its body — the *singe abstract method* of their type — encapsulates the actual logic and evaluates *lazy* at your discretion. They are a simple way to store and transfer expressions for later consumption.

Let's look at some eager code for providing an argument to a method and how it can be made lazy with the help of lambdas.

An Eager Approach

In Example 4-6 a hypothecial `User` is updated with a list of roles. The update isn't always done and depends on the inner logic of the `update` method. The arguments are provided *eagerly*, requiring a pretty expensive lookup call through the DAO[2].

Example 4-6. Updating a user with eager method arguments

```
void updateUser(User user,
                List<Role> availableRoles) { ❶
  // ...
}

// HOW TO USE

var user = loadUserById(23L);

var availableRoles = this.dao.loadAllAvailableRoles(); ❷

var updatedUser = updateUser(user, availableRoles); ❸
```

❶ The `updateUser` method requires the `user` and a list of all available roles. The update itself depends on the inner logic and might not need the roles after all.

❷ The `loadAllAvailableRoles(user)` is done even if the `updateUser` method might not require it, resulting in a costly trip to the database regardless of its necessity.

❸ All arguments are already evaluated at the time of the method call.

Providing `updateUser` with the roles even if they aren't necessary for every use-case wastes performance. So how can you make the call non-mandatory if not absolutely required?

A Lazier Approach

All it needs is the `DAO` to make the call. Let's change the method signature of `updateUser` as seen in Example 4-7.

Example 4-7. Updating a user more lazily

```
User updateUser(User user,
                DAO roleDAO) { ❶
  // ...

  var availableRoles = dao.loadAllAvailableRoles();

  // ...
}
```

The `updateUser` method now has all the tools necessary to load the available roles by itself, if required. But the "solution" to loading the roles only if necessary created another problem: *interdependence*. The `updateUser` method now uses the `DAO` directly and is no longer isolated from *how* the roles are acquired. This approach will make the method *impure* (side-effects from database-access), harder to verify and test (DAO must be mocked). Thanks to possible API boundaries, it gets even more complicated if the `updateUser` method doesn't know about the `DAO` at all. So you need to create another abstract layer to retrieve the roles.

A Lambda Approach

Instead of providing `updateUser` directly with the `DAO`, a lambda will provide the lazy evaluation of loading the roles, as seen in Example 4-8.

Example 4-8. Updating a user with a lambda

```
void updateUser(User user,
                Supplier<<List<Role>> availableRolesFn) { ❶
  // ...

  var availableRoles = availableRolesFn.get();

  // ...
}

// HOW TO USE

var user = loadUserById(23L);

updateUser(user, this.dao::loadAllAvailableRoles);
```

❶ The `updateUser` method signature has to be changed to accept a `Supplier<User, List<Role>>` instead of the already loaded `List<Role>` or the `DAO` itself.

Making `updateUser` a higher-order function by accepting a `Function` creates a superficial new layer without requiring an additional type wrapping the role-loading process. The downsides of using the `DAO` as an argument are all gone:

- No interconnection between the `DAO` and the `updateUser` method, allowing for the possibility of a *pure*, side-effect-free method.
- No additional type as abstraction needed. The already available `Supplier` functional interface provides the simplest level of abstraction possible.
- Testability is restored without requiring the (possibly complicated) mocking of the `DAO`.

By changing the method signature and using one of the already available functional interfaces of the JDK, the `updateUser` method is now able to lazy-load its requirements without the downsides of interconnecting itself to the `DAO`.

Delayed Executions with Thunks

A *thunk* describes an unevaluated expression injected into another subroutine. Thunks fall into the general category of *lazy loading/initialization*, a design pattern often found in object-oriented code. Both are similar mechanisms for achieving the same goal: *non-strict* evaluation. But instead of just deferring the evaluation, like a simple `Supplier`, a thunk also caches the result, so it won't need to re-evaluate it if it's needed again. You've already encountered the concept of storing the result of an evaluated expression in [Link to Come], called *memoization*.

The most straightforward approach is wrapping the `Supplier` in a class, caching the result of the evaluated expression, as shown in Example 4-9.

Example 4-9. Simple Thunk<T>

```
public class Thunk<T> implements Supplier<T> {

    private final Supplier<T> expression; ❶

    private T evaluatedResult; ❷

    public Thunk(Supplier<T> expression) {
        this.expression = expression;
```

```
    }

    @Override
    public T get() {
        if (this.evaluatedResult == null) { ❸
            this.evaluatedResult = this.expression.get();
        }
        return this.evaluatedResult;
    }

    public static <T> Thunk<T> of(Supplier<T> expression) { ❹
        return new Thunk<T>(expression);
    }
}

// Thunk for the previous example

Thunk<List<Role>> availableRoles =
Thunk.of(this::loadAllAvailableRoles); ❺
```

❶ The expression needs to be stored to delay evaluation.

❷ The result must be stored after evaluation.

❸ If not evaluated yet, the expression gets resolved, and the result is stored.

❹ A convenience method to create a `Thunk` without needing `new` or generic type information.

❺ With the help of the `Thunk`, the previous example can be used in a loop without constant re-evaluation.

The thunk implementation follows the *virtual proxy* design-pattern[3]. It mimics a `Supplier<T>` by implementing the functional interface and is, therefore, a drop-in replacement for any `Supplier<T>`, although in an unevaluated form.

Lazy-by-Design

At its core, the idea of laziness boils down to this: deferring the required work to the point it's indispensable. This requires giving up a certain degree of control of the exact time of evaluation. This perceived *loss of control* is where a functional approach really shines. It's rather difficult to predict in what order the code executes, or even at all! That's why functional techniques, like side-effect-free *pure* functions, or *immutable* data structures, and *higher-order* functions, can easily be seen as a precursor to utilize the potential of a lazy approach fully.

One drawback, though, lies in its very nature. Eager evaluation allows for quite linear and compositional performance assessment: the *perceived*, and the *total* performance requirement is the sum of all evaluated parts. Laziness shifts the actual computational cost from where expressions are defined to when they are used, with the possibility of code not being run at all. That's why idiomatic *lazy* performance is harder to assess. The *perceived* performance would most likely improve immediately compared to *eager* evaluation. And the *total* performance requirements may vary on the context and the actual run code.

Software development is a constant battle of utilizing scarce resources most efficiently to reach the desired, or required, performance. Lazy techniques, like delayed evaluation, or Streams for data processing, are a low-hanging fruit[4] to improve your code's performance. It definitely will reduce the required work to a minimum, maybe even zero, freeing up precious performance for other tasks. If some expression or costly computation can be avoided, making it *lazy* will most definitely be a worthwhile endeavor in the long run.

Takeaways

- *Strictness* is about *doing things*; *laziness* is about *considering things to do*.
- *Strict* evaluation means expressions and method arguments evaluate immediately.
- *Lazy* evaluation means deferring the evaluation to the point where it's necessary, maybe even not evaluating at all.

- Java is a *strict* language regarding expressions and method arguments, although certain *lazy* operators and control structures exist.
- Lambas allow the encapsulation of expressions, making them a way to define *lazy* code to be executed at your discretion.
- Streams are *lazy* functional pipelines, and `Optional` and `Map` provide *lazy* additions to their general interfaces.
- The `Supplier` interface is the easiest way to create a *lazy* expression.
- Memoization, like a `Thunk`, helps to avoid re-evaluation.
- *Laziness* is a performance optimization powerhouse: the best code is the one that's not run at all. The next best alternative is to run it *on-demand*-only.
- The assessment of performance requirements for *lazy* code is tricky, though. *Lazy* evaluation shifts the *perceived* and *absolute* performance between *where* expressions are defined and *when* they are used.

---

1 See "Effectively final" for the definition and requirements of *effectively* `final` variables.

2 A *DAO* (data access object) is a pattern to provide an abstract interface to a persistence layer like a database. It translates application calls to specific operations on the underlying persistence layer without exposing details of it.

3 Wikipedia entry on proxies provides an overview over the different kinds of proxies and their usage.

4 The concept of a *low-hanging fruit* describes a goal that is easy to achieve or taken advantage of, compared to the alternatives.

Chapter 5. Optionals

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

Many people call the `null` reference a *billion-dollar mistake*. The inventor of `null` itself originally coined that phrase:

> *I call it my billion-dollar mistake.*
>
> *It was the invention of the* `null` *reference in 1965. At that time, I was designing the first comprehensive type system for references in an object-oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a* `null` *reference simply because it was so easy to implement.*
>
> *This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*
>
> —Sir Charles Antony Richard Hoare, QCon London 2009

There is no consensus on how to deal with this "mistake" among programming languages. But many of them have a proper and idiomatic way of handling `null` references, directly integrated into the language itself.

This chapter will show you the different ways to handle `null` and how you can improve handling `null` in Java with the `Optional<T>` type and its functional API. You will create your own improved Optional type for primitives and learn how, when, and when not to use Optionals.

How to Handle null

Instead of diving into Java's `null`-handling head-first, it's worth looking at how different programming languages handle `null` first. The range of how deep such handling integrates directly into a language varies greatly, and there is no consensus on what's best. But most languages use one of these five concepts to deal with `null` references:

- Best practices and informal rules
- Safe navigation operator
- Embrace `null` as a valid value
- Third-party tools
- No `null` at all and/or specialized types

Best Practices and Informal Rules

If a language itself isn't providing any form of integrated `null` handling, you can only resort to *best practices* and *informal rules*. Many companies, teams, and projects develop their own coding style or adapt an existing one to their needs By adhering to these self-imposed practices and rules, they're able to write more predictable code consistently.

The following four rules are a good starting point for handling `null` references:

- Don't initialize a variable to `null`.

- Don’t pass, accept, or return `null`.
- `null` is acceptable as an implementation detail.
- `null`-check everything outside your control.

These rules aim at reducing the general use of `null`. But `null` isn’t inherently bad. Less usage leads to fewer `null`-checks and `NullPointerExceptions`.

This manual approach is only as good as the level to which everyone adheres to it. And you can only stick to these rules in your own code. If you’re accustomed never to encounter `null`, you might get an unwelcome surprise when dealing with external code. That doesn’t mean that following *best practices* and *informal rules* isn’t a good idea per se. They will improve your overall code quality, regardless of `null`. But it’s not a silver bullet and requires discipline among your team to gain the most benefits from them.

Safe Navigation Operator

Languages with *nullable types* allow a type definition to express that its corresponding value might be `null` and requires special handling, often signified with a "`?`" (question mark) in the type declaration. That leads to the antagonistic requirement that non-nullable types are not allowed to be `null`.

> **NOTE**
>
> Not all languages call their `null` reference `null`. Many languages, like Swift, call it `nil`, but it’s effectively the same thing.

To safely call — or “navigate" — properties or functions of nullable types, a safe navigation operator is used to forgo any `null`-checks. For example, *Swift*, like many other languages with nullable types, uses `?.` as its safe navigation operator, in addition to the single dot you're used to in Java. The code in Example 5-1 shows how it's used for multiple properties. The resulting `name` might be `null` if any of the intermediate calls encountered a `null`-value, or the author's name is `null`. This way, no `null`-checks are required between the calls.

Example 5-1. Safe navigation operator in Swift

```
// nil is Swift's literal for a null reference

let name: String? = articles?.first?.author?.name
```

Java requires a lot more checks to be as safe as its equivalent Swift version. You have to `null`-check every single step, as seen in Example 5-2 to avoid any `NullPointerExceptions`.

Example 5-2. Unsafe navigation with Java

```
String name = null;

if (articles != null && articles.isEmpty() == false) {

   var article = articles.get(0);

   // Additional check needed if collection allows null values
   if (article != null) {

        var author = article.getAuthor();

        if (author != null) {
            name = author.getName();
        }
    }
 }
```

The extra boilerplate makes your code neither concise nor easy to reason with. And to make things worse, anyone using the variable `name` can't deduce that it might be `null`, because its type definition — `String` — doesn't confer the valid values.

Null Coalesce

Swift also provides another part of the equation for better `null` handling: a *null coalesce operator*.

The safe navigation operator makes handling possible `null` values easier. But at some point, you might need a fallback if it's `null`. That's where another operator comes into play, the `"??"` (double question mark) operator.

If the left-side of the operator is `null`, the right side evaluates, as seen in Example 5-3.

Example 5-3. Swift null coalesce operator

```
let name: String = articles?.first?.author?.name ?? "n/a"
```

This operator is like a short form of `"? :`", the ternary operator. Example 5-4 shows the null coalesce operator and how to implement the same logic with the ternary operator.

Example 5-4. Swift null coalesce operator versus ternary operator

```
let name: String = articles?.first?.author?.name ?? "n/a"

// is equivalent to

let maybeName: String? = articles?.first?.author?.name
let name: String = maybeName != nil ? maybeName! : "n/a"
```

The difference between the two versions is the evaluation order and count. The null coalesce operator is a binary operator, which means that if the left side evaluates to non-`null`, the expression on the right side won't evaluate at all. And its operators evaluate only once at most. That is an essential trait if one of the operands is an expression or method call with side effects.

As with the safe navigation operator, Java doesn't have a null coalesce operator. But the `Optional` type provides an alternative way to navigate potential `null` properties and methods, as you will learn later in this chapter.

Embrace null as valid value

Another approach is not throwing exceptions on `null` references and accepting it as a valid value.

Clojure, a functional programming language running on the JVM, made the *absence* of a value an acceptable state. Even though its `nil` is analogous to Java's `null`, it's evaluating to `false`. It's still possible to encounter a `NullPointerException`, especially in Java-interop code. But it's way more complicated in idiomatic *Clojure* code. With `nil` being a value representing *nothingness* instead of *being nothing* itself, it's way easier to handle and incorporate it into your code.

Another example is *Objective-C*, a *Smalltalk*-inspired language. Semantically speaking, it doesn't *call* methods or fields. Instead, it *sends messages* to objects and might receive a response. Sending a message to `nil` will not raise an exception. Instead, the message will be discarded silently. A language with manually managed memory, like *Objective-C*, can benefit from this behavior because you don't have to `null`-check everything before sending a message. But it's also bad because you might not realize that a sent message got discarded, and now you might have to check for that instead.

Third-party tools

If a language doesn't provide the built-in functionality you need, you can always augment it with third-party tools to provide the missing features. For `null` references in Java, an established *best-practice* is to use annotations to mark variables, arguments, and method return types as either `@Nullable` or `@NonNull`. It allows `static` code analysis to find possible problems with `null` at compile-time. And even better, adding these annotations to your code gives your method signatures and type definitions a clearer intent of how to use them and what to expect, as seen in Example 5-5.

Example 5-5. Null handling with annotation

```
interface Example {

  @NonNull List<String> getListOfNullables(); ❶

  List<@NonNull String> getNullableListOfStrings(); ❷
```

```java
  void doWork(@Nullable String identifier); ❸
}
```

❶ Returns a non-null List of possible null String objects.
❷ Returns a possible null List containing non-null String objects.
❸ The method argument identifier is allowed to be null.

The JDK doesn't include these annotations, and the corresponding JSR 305 has the status "dormant". But it's still the *de-facto* community standard. Several libraries[1] provide the missing annotations, and most tools support multiple variants of them. But be aware that the tools might differ in their actual handling of certain edge-cases[2].

The general problem with a tool-assisted approach is the reliance on the tool itself. If it's too intrusive, you might end up with code that won't run without it, especially if the tool involves code generation "behind the scenes". In the case of null-handling, that isn't a big worry. Your code will even run without a tool interpreting the annotations, and your variables and methods signatures will still clearly communicate the requirements to anyone using them, even if unenforced.

No null at all and/or specialized types

The most *drastic* method for dealing with null is not allowing it at all. For example, the functional language *Haskell* doesn't have null as a design feature. Although the lack of any null-related exceptions sounds excellent, you still need to somehow deal with the general concept of the *absence* of a value. Many languages, including *Haskell*, solve this by providing specialized types for representing optional values. Their names are almost always in the same vein: Option, Optional, Maybe, Some, etc. They allow for excellent null-handling without requiring an explicit language or syntax integration.

Java's Optionals are such a specialized type.

How Java Handles null with Optionals

Dealing with `null` in Java can be cumbersome. Correct handling of `null` is an essential part of every developer's work because unexpected and unhandled `NullPointerExceptions` are the root cause of many problems with software written in Java. Without any language-integrated features, like a safe navigation or null coalesce operator, you must resort to best practices or third-party tools. Adhering to best practices is always a good idea, although it's hard to enforce them strictly. Instead, a tool-assisted approach might be easier to enforce but can complicate the general project workflow.

Java 8 finally introduced a specialized type to handle `null`: `java.util.Optional<T>`. As with many other Java 8 features, lambdas are part of the general design of Optionals, so a more functional programming style becomes feasible and beneficial by introducing more concise ways of handling nullable values.

Optional<T> Operations

The `Optional<T>` type is not just a simple wrapper around a value. It provides almost 20 methods in four different categories:

Creating an Optional

> The first step is creating an Optional. It can either have a value or is empty.

Are you there, value?

> You can check for the existence of an inner value with `isEmpty()` or `isPresent()`, with both returning a `boolean`. Alternatively, there are two methods available for a functional approach: `ifPresent(Consumer<? super T> action)` or `ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`.

Filtering and Mapping

Like with Streams, you use these *intermediate operations* to work with the value in an Optional by filtering or transforming it. Each operation returns an Optional containing the resulting value, or an empty Optional if no value is present. You can connect the method calls fluently to another *intermediate* or *terminal* operation.

Getting the value, or having a backup plan

The `Optional<T>` type comes with multiple `or…(…)` methods. In case of an empty Optional, you can provide an alternative by using one of them. Many use-cases are covered, like eager and lazy values, exceptions, and even new Optionals.

Creating Optionals

Optionals do not provide any `public` constructor. Instead, three `static` methods are available for three distinct use cases:

Case 1: There might be a value

The main intention of Optionals is to make dealing with `null` more of a non-issue. If you simply need an Optional and don't care if it might be empty, use the method `Optional.ofNullable(…)`. It's the simplest and most bullet-proof form of creating an Optional.

```
var value = "Optionals are awesome!";
Optional<String> maybeValue = Optional.ofNullable(value);

value = null;
Optional<String> emptyOptional = Optional.ofNullable(value);
```

Case 2: Value is known/needed

Even though Optionals are a great way to deal with `null` and prevent a `NullPointerException`, what if you have to make sure you have a value? For example, you already handled any edge-cases in your code

— which returned empty Optionals — and now you definitely have a value. The static method `Optional.of(…)` ensures that the value is non-`null`, and throws an `NullPointerException` if not. This way, the exception signifies a real problem in your code. Maybe you missed an edge case, or a particular external method call has changed and returns `null` now. Using `Optional.of(…)` in such a context makes your code more future-proof and resilient.

```
var value = "Optionals are awesome!";
Optional<String> mustHaveValue = Optional.of(value);

value = null;
Optional<String> emptyOptional = Optional.of(value);
// => throws NullPointerException
```

Case 3: There's never a value

If you know there's no value in the Optional, you can use the method `Optional.empty()`. That is especially useful for return values because you don't have to create a new empty `Optional` every time. `Optional.empty()` is a `static final` field, and is also returned by `Optional.ofNullable(null)`.

```
Optional<String> noValue = Optional.empty();
```

Checking for Values

The primary purpose of Optionals is to wrap a value and to represent its existence or absence. So naturally, checking for values must be as straightforward as possible. There are four methods available for checking and reacting to values or their absence. They fall into two sub-categories, the `is` methods and the `if` methods.

`boolean isPresent()`

Returns `true` if a value is present.

`boolean isEmpty()`

Returns `true` if the Optional is empty. This method was added with Java 11, so you don't have to check `!isPresent()`, making your code more readable.

Instead of checking, retrieving, and using a value in separate steps, you could use one of the `if` methods instead to streamline your code.

`void ifPresent(Consumer<? super T> action)`

Performs the supplied action only if a value is present. `null` actions aren't allowed and throw a `NullPointerException`.

`void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`

Performs the supplied action only if a value is present. If no value is found, the `emptyAction` is run instead. `null` actions aren't allowed and throw a `NullPointerException`.

Let's look at how to use these methods in Example 5-6.

Example 5-6. Checking for Optional values

```
Optional<String> maybeValue = ...;

// THIS VERBOSE CODE...

if (maybeValue.isPresent()) {
  var value = maybeValue.get();
  System.out.println(value);
}
else {
  System.out.println("No value found!");
}

// ...CAN BE SIMPLIFIED TO

maybeValue.ifPresentOrElse(System.out::println,
                           () -> System.out.println("No value
found!"));
```

Filtering and Mapping

Optionals allow for more than just checking for the presence of a value or its absence. Similar to Streams, you can build a pipeline by filtering and mapping values. Every step results in a new Optional until a terminal operation ends the call-chain.

`Optional<T> filter(Predicate<? super T> predicate)`

Filter a value with the provided predicate. Returns `this` if no value is present because it's already an empty Optional.

`<U> Optional<U> map(Function<? super T,? extends U> mapper)`

Transforms a value with the provided mapper function, returning a new nullable Optional containing the mapped value. If no value is present, an empty `Optional<U>` is returned instead.

`<U> Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper)`

If your mapping function returns an `Optional<U>` instead of a concrete value of type `U`, using `map(…)` would result in an `Optional<Optional<U>>`. But `flatMap(…)` doesn't pack the result of the mapper function into a new Optional.

These intermediate operations allow you to build a call-chain to filter and transform a value as needed. Example 5-7 shows an Optional call-chain and the non-Optional equivalent for a hypothetical permissions container and its sub-types. The code callouts are attached to both versions to show the corresponding operations, but their descriptions are for the Optional-version.

Example 5-7. Intermediate operations

```
interface Permissions {
  boolean isEmpty()
  Group getGroup();
```

```
}

interface Group {
  Optional<User> getAdmin();
}

interface User {
  boolean isActive();
}

Permissions permissions = ...;

// WITH OPTIONALS

boolean isActiveAdmin =
  Optional.ofNullable(permissions) ❶
          .filter(Predicate.not(Permissions::isEmpty)) ❷
          .map(Permissions::getGroup) ❸
          .flatMap(Group::getAdmin) ❹
          .filter(User::isActive) ❺
          .orElse(Boolean.FALSE); ❻


// WITHOUT OPTIONALS

boolean isActiveAdmin = false; ❻

if (permission != null && !permissions.isEmpty()) { ❶ ❷
  if (permission.getGroup() != null) { ❸
    var group = permissions.getGroup(); ❸
    var maybeAdmin = group.getAdmin(); ❹

    if (maybeAdmin.isPresent()) {
      var admin = maybeAdmin.get();
      isActiveAdmin = admin.isActive();
    }
  }
}
```

❶ Initial `null`-check by creating an `Optional<Permissions>`.

❷ Filter for non-empty permissions, using a `static` helper of the `Predicate` type to simplify the lambda to a method reference.

❸ Get the group of the permissions. It doesn’t matter if `getGroup()` returns `null`, because the `Optional` call-chain will skip to its terminal operation if that’s the case. The non-Optional version needs an explicit `null`-check if you can’t guarantee that group is never `null`.

The group might not have an admin. That’s why it returns an

❹ The group might not have an admin. That's why it returns an Optional<Admin>. If you simply use map(Group::getAdmin), you would have an Optional<Optional<Admin>> in the next step. Thanks to flatMap(Group::getAdmin), the unnecessarily nested Optional won't be created.

❺ With the Admin object, you can filter out non-active ones.

❻ If any method of the call-chain returns an empty Optional, e.g., the group was null, the terminal operation returns the fallback value Boolean.FALSE. The next chapter will explain the different types of terminal operations.

The difference between the two versions is quite noticeable. The Optional call-chain is a fluent call that's almost prose-like. Every step of the underlying problem that needs to be solved is laid out in clear and directly connected steps. Like null or empty-checks, any validation and decision-making are wrapped up in the Optionals operations and method references. The intent and flow of the problem to be solved are clearly visible, even without explicit if-statements.

The non-Optional version can't delegate any conditions or checks and relies on explicit if-statements. That creates deeply nested flow structures, increasing the *cyclomatic complexity* of your code. It's harder to understand the overall intent of the code-block and not as concise as with an Optional call-chain.

> **NOTE**
>
> *Cyclomatic Complexity*[3] is a metric used to determine code complexity. It's based on the number of branching paths — or decisions — in your code. The general idea is that straight, non-nested statements and expressions are more accessible to follow than deeply nested decision branches, like nested if-statements.

Getting a (fallback) value

The simplest way of retrieving the inner value is calling `get()`. But make sure you've checked for the existence of a value beforehand, or you'll end up with a `NoSuchElementException`. Instead of using `get()`, you can use the more flexible `or`-prefixed methods, giving you a chance to define an alternative if no value is present.

`T orElse(T other)`

Returns either the value of the Optional or "`other`" if no value is present.

`T orElseGet(Supplier<? extends T> supplier)`

Instead of needing an alternative right away, you can supply it lazily with a `Supplier`.

`<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`

Even though one of the main advantages of Optionals is preventing `NullPointerException`, sometimes you still need an exception if there's no value present. With `orElseThrow(…)`, you have fine-grained control about handling a missing value and what exception to throw, too.

`T orElseThrow()`

Like `orElseThrow(…)`, but in case no additional handling or specific exception type is needed. Always throws a `NoSuchElementException` if no value is present.

There's another method available since Java 9. Even though it's not a *terminal* operation providing you with a tangible value, it helps build more complex Optional call-chains.

`Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)`

Lazily return another `Optional<T>` if no value is present.

Not fitting in the *or* nomenclature is another value-providing method used for Stream pipelines.

`Stream<T> stream()`

Returns a `Stream<T>` only containing the value, or an empty Stream if not value is present. Often used in `Stream#flatMap()` as a method reference.

Optional Primitives

As you have learned about functional interfaces and Streams, handling primitives requires additional types in Java. `Optional<T>` is a generic type so it can't handle primitives, therefore you need specialized types for primitives, too.

You might ask yourself why you might even need an Optional of a primitive: primitives can never be `null`! That's correct. But Optionals aren't only about preventing values to be `null`. They're also able to represent a state of nothingness — an absence of a value — and not an always identical primitive default value. In many cases, these default values are enough, like representing a networking port. Zero is an invalid port number, so you have to deal with it anyway.

There are two options available so far to deal with primitives: auto-boxing or specialized types. "Primitive Types" highlighted the problems of using object-wrapper classes and the overhead they introduce. On the other hand, auto-boxing isn't free either.

The usual primitive types are available as specialized Optionals:

- `java.util.OptionalInt`

- `java.util.OptionalLong`
- `java.util.OptionalDouble`

Their semantics are almost equal to their generic counterpart, but they do **not** inherit for `Optional<T>` or share a common interface. The features aren't identical either, multiple methods, like `filter(…)`, `map(…)`, or `flatMap(…)`, are missing.

The specialized primitive Optional types remove unnecessary auto-boxing. But with the lack of some of the functionality provided by `Optional<T>`, you might resort to using the generic wrapper more often than you want to. Instead of accepting the status-quo, why not create a better alternative yourself?

Creating an Improved Primitive Wrapper

With the specialized types not being directly related to their generic counterpart, you can write your own improved Optional types, including all the missing features and more. Let's create an improved version of `OptionalInt`, including the missing methods and even some new functionality to allow interoperability between `OptionalInt` and `Optional<T>`.

Instead of replicating the full functionality of `OptionalInt`, the new type uses the *delegation pattern* to achieve the same code-reuse as inheritance, without actually inheriting from `OptionalInt`, which can't be inherited.

> **NOTE**
>
> The *delegation pattern* is an object-oriented design pattern based on object composition. It follows the "composition over inheritance" design principle, allowing for code-reuse without explicitly inheriting from other types[4].

Example 5-8 is the minimal implementation of replicating the functionality of `OptionalInt`. All methods available on `OptionalInt` are delegating their work, so most of the "delegation-only" methods are omitted for readability.

Example 5-8. Improved OptionalInt (minimal functionality)

```
public class ImprovedOptionalInt {

  private final OptionalInt delegate; ❶

  private static final ImprovedOptionalInt EMPTY = new
ImprovedOptionalInt();

  public static ImprovedOptionalInt empty() { ❷
    return EMPTY;
  }

  public static ImprovedOptionalInt of(int value) { ❷
    return new ImprovedOptionalInt(value);
  }

  private ImprovedOptionalInt(int value) {
    this.delegate = OptionalInt.of(value);
  }

  private ImprovedOptionalInt() {
    this.delegate = OptionalInt.empty();
  }

  public boolean isPresent() { ❸
    return this.delegate.isPresent();
  }

  // various delegation-only methods omitted for readability

  @Override
  public boolean equals(Object obj) { ❹
    if (this == obj) {
      return true;
    }

    if (!(obj instanceof ImprovedOptionalInt)) {
      return false;
    }

    var other = (ImprovedOptionalInt) obj;
    return this.delegate.equals(other.delegate);
  }
```

```
  @Override
  public int hashCode() { ❸
    return this.delegate.hashCode();
  }

  @Override
  public String toString() { ❺
    return isPresent() ? String.format("ImprovedOptionalInt[%d]",
this.delegate.getAsInt())
                       : "ImprovedOptionalInt.empty";
  }
}
```

❶ The `delegate` that does the "heavy lifting" for the improved type.

❷ The same convenience creation methods as with `OptionalInt`

❸ If no additional logic is required — as for most methods in the type — a simple delegate call is sufficient.

❹ The `equals` method has a slightly different implementation to check against the new type.

❺ The `toString` method is also adapted to use the new type name.

If you include all the delegated methods, the new type contains quite a lot of code. But thanks to the delegation pattern, it's mostly a single call per method.

Now that we got feature-parity with `OptionalInt`, it's time to add the missing methods from `Optional<T>`, as seen in Example 5-9.

Example 5-9. Improved OptionalInt (missing methods)

```
public ImprovedOptionalInt filter(IntPredicate predicate) {

  Objects.requireNonNull(predicate);
  if (isEmpty()) {
    return this;
  }

  return predicate.test(getAsInt()) ? this : empty();
}

public <U> Optional<U> map(IntFunction<? extends U> mapper) {

  Objects.requireNonNull(mapper);
  if (isEmpty()) {
    return Optional.empty();
```

```
    }

    return Optional.ofNullable(mapper.apply(getAsInt()));
  }

  public <U> Optional<U> flatMap(IntFunction<? extends Optional<?
  extends U>> mapper) {

    Objects.requireNonNull(mapper);
    if (isEmpty()) {
      return Optional.empty();
    }

    @SuppressWarnings("unchecked")
    var value = (Optional<U>) mapper.apply(getAsInt());
    return Objects.requireNonNull(value);
  }

  public ImprovedOptionalInt or(Supplier<? extends
  ImprovedOptionalInt> supplier) {
    Objects.requireNonNull(supplier);
    if (isPresent()) {
        return this;
    }

    return Objects.requireNonNull(supplier.get());
  }
```

By supporting the intermediate methods `filter(…)`, `map(…)`, and `flatMap(…)`, the new type is en par with `Optional<T>` in regard to the available functionality. The implementations of these methods are based on their equivalent methods of `Optional<T>`. But they use their respective specialized functional interfaces to avoid as much auto-boxing as possible.

The `ImprovedOptionalInt` is already more versatile than a plain `OptionalInt`. But you can improve it even further. When you design new types, it’s always a good idea to think ahead. By that, I don’t mean to over-engineer your types to anticipate any possible future edge-case before actually needing it. But interoperability with existing types is a good addition, as seen in Example 5-10.

Example 5-10. Improved `OptionalInt` (interop)

```
public Optional<Integer> boxed() {

  if (isEmpty()) {
    return Optional.empty();
  }

  Integer boxedValue = Integer.valueOf(getAsInt());
  return Optional.ofNullable(boxedValue);
}

public Optional<Integer> boxedStream() {

  if (isEmpty()) {
    return Stream.empty();
  }

  Integer boxedValue = Integer.valueOf(getAsInt());
  return Stream.of(boxedValue);
}

public OptionalInt optionalInt() {
  return this.delegate;
}
```

With these three additions, the `ImprovedOptionalInt` provides compatibility with its specialized counterpart `OptionalInt` and the boxed variant `Optional<Integer>`.

Using Your Own Optional Types

Even though we developed an improved Optional type for `integer` primitives in the previous section, I actually wouldn't recommend using it under most circumstances. Internally, you can use whatever Optional type you prefer. But for a public API, you should always strive to use the most anticipated type, and that's usually what's already included in the JDK.

The lesson here was to show you that Optionals aren't magic types. You can easily create your own, either by delegating the actual work or reimplementing the logic yourself. If you look at the actual implementation of any method in `OptionalInt`, like shown in Example 5-11, you see that it's mostly "boilerplate" code to handle the absence and presence of a value.

Example 5-11. OptionalInt#ifPresentOrElse implementation

```
public void ifPresentOrElse(IntConsumer action, Runnable
emptyAction) {
  if (isPresent) {
    action.accept(value);
  } else {
    emptyAction.run();
  }
}
```

Optionals and Streams

Optionals alone are already a great addition to the JDK. Their fluent functional API allows you to reduce the usual boilerplate code needed for typical operations immensely. But they also provide interoperability with another fluent functional API highlighted in this book: Streams.

Optionals as Stream Elements

Streams are pipelines working on elements, filtering and transforming them to a desired outcome. To fit into such pipelines, Optionals can be seen as a filter operation. If a value is present, it should provide it to the Stream. If not, `Stream.empty()` is returned. This behavior is ingrained in the `Optional<T>#stream()` method. Combined with the intermediate Stream operation `flatMap(…)`, you can save a `filter(…)` and `map(…)` operation to check and unpack an Optional, as seen in Example 5-12. The code callouts are attached to both versions to show the corresponding operations, but their descriptions are for the `flatMap`-version.

Example 5-12. Optionals as Stream elements

```
List<Permissions> permissions = ...;

// WITH FLATMAP
```

```
List<User> activeUsers =
  permissions.stream()
             .filter(Predicate.not(Permissions::isEmpty))
             .map(Permissions::getGroup)
             .map(Group::getAdmin) ❶
             .flatMap(Optional::stream) ❷
             .filter(User::isActive)
             .orElse(Collections.emptyList());

// WITHOUT FLATMAP

List<User> activeUsers =
  permissions.stream()
             .filter(Predicate.not(Permissions::isBlock))
             .map(Permissions::getGroup)
             .map(Group::getAdmin) ❶
             .filter(Optional::isPresent) ❷
             .map(Optional::get) ❷
             .filter(User::isActive)
             .orElse(Collections.emptyList());
```

❶ The `getAdmin()` returns an `Optional<Admin>`. At this point, the Stream became a `Stream<Optional<Admin>>`.

❷ Streams already have a concept of the existence and absence of elements in the pipeline. If an element is an Optional, it can't be absent. But the value of the Optional can still be absent, which is why you need to reduce the Stream from `Stream<Optional<Admin>>` to `Stream<Admin>`. That way, the "normal" Stream semantics are restored.

Even though you only save a single method call — `flatMap` instead of `filter` and `map` — the resulting code is easier to reason with. The `flatMap` operation conveys all the necessary information for understanding the Stream pipeline without adding any complexity by being split into multiple steps. Handling Optionals is a necessity, and it should be done as concisely as possible so that the overall Stream pipeline is as understandable and straightforward.

There's no reason to design your APIs without Optionals just to avoid `flatMap` operations in Streams. If `getAdmin()` would return `null`, you would have to check for it with `filter(Objects::nonNull)` anyways. And replacing a `flatMap` operation with a `filter` operation gains you nothing, except `getAdmin()` now requires explicit `null`-handling, even if it's not obvious.

Terminal Operations

Five of the Stream API's terminal operations provide an Optional as their result. All of them can be categorized as trying to find or produce a value. But in the case of an empty Stream, there had to be a sensible representation of an absentee value. With Optionals being an exemplary manifestation of this concept, it was logical to use them instead of return `null`.

Finding an Element

In the Stream API, the prefix `find` represents finding an element based on its existence, compared to `match`, which embraces the element's state in its decision. There are two `find` operations available with different semantics on the encountered order:

`Optional<T> findFirst()`

Returns the first element of a Stream, or an empty Optional if the Stream is empty. Any element might be returned if the Stream lacks an encounter order. See [Link to Come] for more details on Stream characteristics.

`Optional<T> findAny()`

Return any element of a Stream, or an empty Optional if the Stream is empty. The returned element is non-deterministic to maximize performance in parallel streams. If you need a consistent return element, you should prefer `findFirst()`.

Reducing to a Single Value

Reducing elements of a Stream into a new data structure is one of its main purposes. And just like the `find` operations, reducing operators have to deal with empty Streams.

Optional<T> min(Comparator<? super T> comparator)

Return the “minimum” element based on the provided comparator, or an empty Optional if the Stream is empty.

Optional<T> max(Comparator<? super T> comparator)

Return the “maximum” element based on the provided comparator, or an empty Optional if the Stream is empty.

Optional<T> reduce(BinaryOperator<T> accumulator)

Reduces the elements of the Stream using the accumulator operator. The returned value is the result of the reduction, or an empty Optional if the Stream is empty. See Example 5-13 for an equivalent pseudo-code example from the official documentation[5].

Example 5-13. Pseudo-code equivalent to reduce (BinaryOperator<T> accumulator)

```
boolean foundAny = false;
T result = null;

for (T element : <this stream>) {
  if (!foundAny) {
    foundAny = true;
    result = element;
  }
  else {
    result = accumulator.apply(result, element);
  }
}

return foundAny ? Optional.of(result)
                : Optional.empty();
```

See [Link to Come] for a more detailed explanation of reduction.

Caveats

There are still caveats with the Optional types due to them being "normal" types, like any other type in the JDK. Any reference to an Optional itself can be `null`, with all its associated problems. If you design an API and decide to use Optionals, you **must not** return `null` under any circumstances! Always use `Optional.empty()` or the primitive equivalent instead. This essential design requirement has to be enforced by convention, though. The compiler won't help you there without additional tools, like for example [The SonarSource[6].

Even though Optionals are "normal" types, certain "taken for granted" features might work differently from other objects. The identity-sensitive operations `equals(…)` and `hashCode` are based on the value within. The results of these methods are described as unpredictable in the official documentation and should be avoided.

Another point to consider is the performance implication of Optionals. Every method call creates a new stack frame, and can't be as easily optimized by the JVM as a `if`-statement of a `null`-check. But usually, the trade-off between performance and safer and more straightforward code tends to be in favor of the latter. Saving a few CPU cycles means nothing compared to a crash due to a unexpected `NullPointerException`.

Special Considerations for Collections

There's another type of data structure that can represent the absence of a value: collections. That's why you shouldn't wrap any collection type, like `List<T>`, in an Optional. If no value is present, use an empty collection instead. It's unclear what an empty Optional of a collection is supposed to represent. Because a collection can already represent an empty state by itself, one might guess an empty Optional signifies the inability to gather any values at all, not just the absence of values. But without additional context or comments, you can't be sure. That's why you should use

Optionals to represent the absence of values for types that already have a concept of absence and existence of values.

If you still need to represent additional states, you have two options. You can either throw an appropriate exception. Or return another type being capable of representing multiple states instead.

Alternative Implementations

Guava[7], the popular "Google core libraries for Java", provides its own `Optional<T>` type since 2011, three years before the release of Java 8. The general semantics are quite similar, but they differ in three aspects:

- Guava's `Optional<T>` implements `Serializable`.
- Java's `Optional<T>` has additional methods like `ifPresent(…)`, `filter(…)`, and `flatMap(…)`.
- Guava doesn't offer specialized types for primitives.

Even though there's now an alternative available directly in the JDK, Guava doesn't plan to deprecate the class in the foreseeable future. If you're already heavily invested in Guava, it makes perfect sense to keep Guava's Optional and not replace everything with the JDK version. Especially since conversion between the two types is simple, as shown in Example 5-14.

Example 5-14. Convert between Guava and JDK `Optional<T>`

```
public final class Optionals {

  public static <T> java.util.Optional<T>
toJDK(com.google.common.base.Optional<T> op) {
    return op.isPresent() ? java.util.Optional.of(op.get())
                          : java.util.Optional.empty();
  }

  public static <T> com.google.common.base.Optional<T>
toGuava(java.util.Optional<T> op) {
    return op.isPresent() ?
com.boogle.common.base.Optional.of(op.get())
                          :
```

```
com.boogle.common.base.Optional.empty();
  }
}
```

Is `null` Really Evil?

Although it's called a *billion-dollar mistake*, `null` isn't inherently evil. Sir Charles Antony Richard Hoare, the inventor of `null`, believes that programming language designers should be responsible for errors in programs written in their language. A language should provide a solid foundation with a good deal of ingenuity and control. Allowing `null` references is one of many design choices for Java. Java's *catch or specify requirement* and `try-catch`-blocks provide you with tools against obvious errors. But with `null` being a valid value for any type, every reference is a possible crash waiting to happen. Even if you think something can *never* be `null`, experience tells us that it will be at some point in time.

These downsides to `null` references don't make Java a poorly designed language. `null` has its place, but it requires you to be more attentive about your code. And it doesn't mean you should replace every single variable and argument in your code with Optionals. Especially in code under your control, you can make more assumptions and guarantees about the possible nullability of references and deal with it accordingly. If you follow the other principles highlighted in this book — like small, self-contained, pure functions without side effects — it's way easier to make sure your code won't return a `null` reference unexpectedly.

Takeaways

- There's no language-level or special syntax available for `null` handling in Java.
- The `Optional<T>` type allows for dedicated `null`-handling with operation chains and fallbacks.

- Specialized types for primitives are also available, although they don't provide feature-parity.
- Other approaches for `null`-handling exist, like annotations, and are de-facto standards.
- Not everything is a good fit for Optionals. If a data structure already has a concept of emptiness, like collections, no additional wrapper is needed.
- Optionals and Streams are interoperable without much friction.
- Alternative implementations exist, like Guava.
- `null` isn't evil per se. Don't replace every variable with Optionals without a good reason.

---

1 The most common libraries to provide the marker annotation are FindBugz (up to Java 8), and its spiritual successor SpotBugz. JetBrains, the creator of the IntelliJ IDE and the JVM language *Kotlin*, also provide a package containing the annotations.

2 The Checker Framework has an example of such "non-standard" behavior between different tools.

3 McCabe, TJ. 1976. "A Complexity Measure" IEEE Transactions on Software Engineering, December 1976, Vol. SE-2 No. 4, 308–320.

4 More information about the delegation pattern can be found in the "Gang of Four" book about design patterns. (Gamma, Erich, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 2016. ISBN 0-201-63361-2.)

5 Documentation for `Optional<T> reduce(BinaryOperator<T> accumulator)`.

6 The SonarSource rule RSPEC-2789 checks for Optionals being `null`.

7 Guava: Google Core Libraries for Java GitHub page.

Chapter 6. Recursion

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

Many developers see recursion as just another — more difficult — approach to iteration-based problem-solving. But it's also a whole different philosophy for solving a particular group of problems in a functional way. If a problem can be broken down into smaller versions of itself, *recursion* can be a better approach, even if it almost seems contradictory to solve a problem with the result of a smaller version of the same problem.

This chapter shows how to implement recursive methods in Java and their implications compared to other forms of iteration.

Mathematical Explanation

In "Recursion", you've seen the example of calculating factorials — the product of all positive integers less than or equal to the input parameter. Many books, guides, and tutorials use factorials because it's a perfect problem to solve partially, and it'll be one of the examples of this chapter, too.

Every step of the calculation breaks down into the product of the input parameter and the next factorial operation. When the calculation reaches `fac(1)` — defined as “1" — it terminates and provides the value to the previous step. The complete steps can be seen in Equation 6-1.

Equation 6-1. Formal representation of factorial calculation

$$
\begin{aligned}
& fac(n) \\
\rightarrow \quad & n * fac(n-1) \\
\rightarrow \quad & n * (n-1) * fac(n-2) \\
\rightarrow \quad & 4 * (n-1) * (n-2) * \cdots * fac(1) \\
\rightarrow \quad & 4 * (n-1) * (n-2) * \cdots * 1
\end{aligned}
$$

This generalization of the calculation steps also shows the general concept of *recursion*. An operation is repeated with different input parameters until it reaches its base condition. Recursion consists of two distinct operation types:

Base conditions

> A base condition is a predefined case — a *solution* to the problem — which will return an actual value and unwind the recursive call-chain. It provides its value to the previous step, which can now calculate a result and return it to its predecessor.

Recursive call

> Until the call-chain reaches a *base condition*, every step will create another step with modified input parameters.

Figure 6-1 shows a more generic flow of a recursive call-chain.

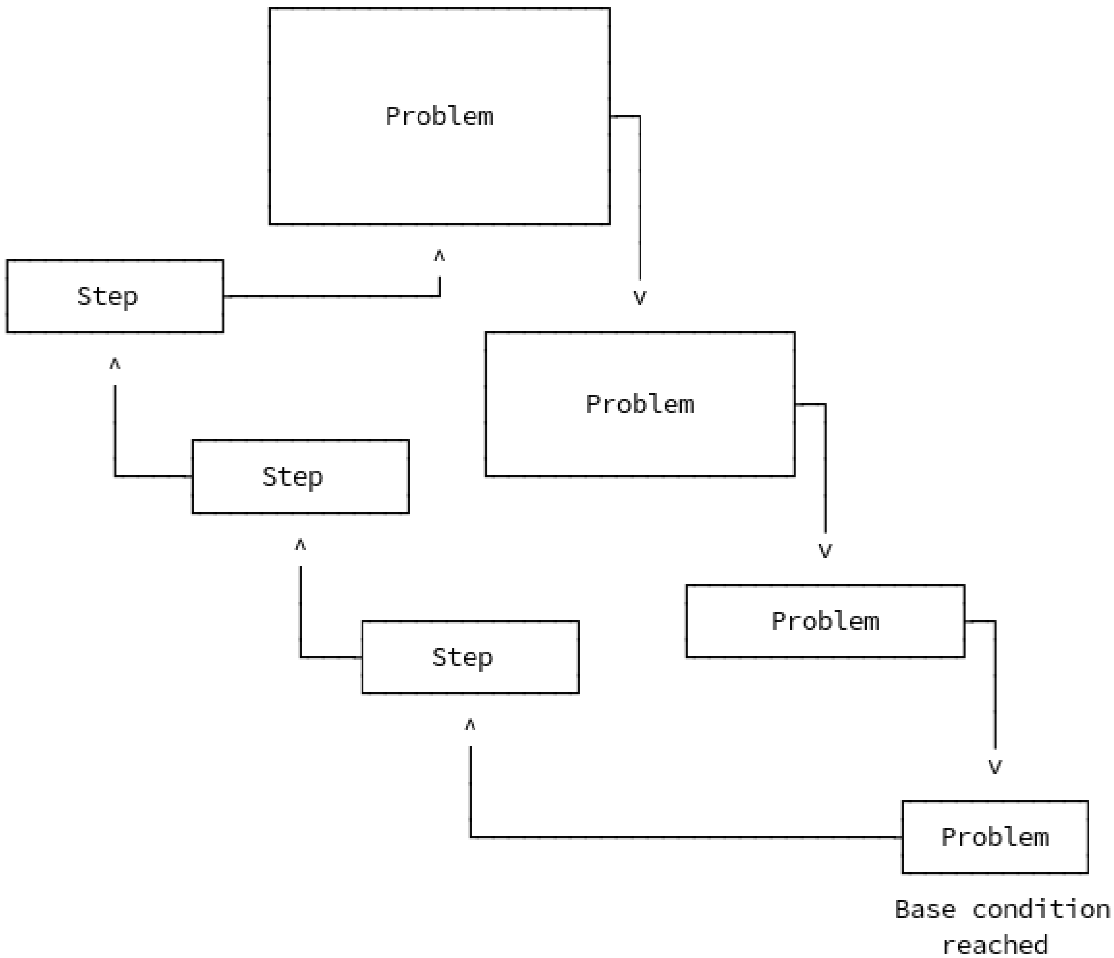

Figure 6-1. Solving problems with smaller problems

Head Versus Tail Recursion

There are two kinds of recursion: *head* and *tail* recursion. The difference is the position of the recursive call:

Head recursion

The recursive call is not the function's last statement, and other statements/expressions are executed/evaluated after it.

Tail recursion

All non-recursive expressions are evaluated before the recursive call.

Let's look at how to calculate a factorial with both types. Example 6-1 shows how to use head recursion.

Example 6-1. Calculating factorials with head recursion

```
long headFactorial(long n) { ❶
  if (n == 1L) { ❷
    return 1L;
  }

  return n * headFactorial(n - 1L); ❸
}
```

❶ The method signature only contains the input parameter of the current recursive step. No intermediate state moves between the recursive calls.

❷ The base condition comes before the recursive call, so the call-chain can finish on fulfilling its condition.

❸ The recursive call depends on the input parameter, and an expression is the actual return value, making it not the last call in the method.

Now it's time to look at tail recursion, as shown in Example 6-2.

Example 6-2. Calculating factorials with tail recursion

```
long factorialTail(long n, long accumulator) { ❶
  if (n == 1L) { ❷
    return accumulator;
  }

  return factorialTail(n - 1L, n * accumulator); ❸
}

factorialTail(4L, 1L); ❹
```

❶ The method signature contains an accumulator.

❷ The base condition hasn't changed compared to head recursion.

❸ Instead of returning an expression dependent on the next recursive call, both `factorialTail(…)` parameters are independent of it. The method only returns the recursive call itself.

❹ The accumulator requires an initial value. It reflects the base condition.

There's one significant advantage to tail recursion. Many modern compilers can use *tail-call optimization/elimination* to optimize the required stack frames. The Java compiler and runtime lack that particular ability, though. That can cause severe problems if the recursive chain-call gets too big. But knowing about the general problem can help you prevent it, as we'll lay it out in the next chapter.

Recursion and the Stack

Every single recursive step is a method call, which creates a new stack frame. That is a necessity because every variable `n` must be isolated from the previous calculation. The recursive call count is only constrained by how long it takes to reach the base condition. The problem is, though, that the available stack size is finite. Too many calls will fill up the available space and lead to a `StackOverflowError`.

If you look at Figure 6-1 again, you can think of every box as a separate stack frame.

> **NOTE**
>
> A stack frame contains the state of a single method invocation. Every time your code calls a method, the JVM creates and pushes a new frame on the global stack. After returning from a method, its stack frame gets popped and discarded.
>
> The actual maximum stack depth depends on the available stack size[1], and what's stored in the individual frames.

Tail recursion and *tail-call optimization* allow compilers to reduce the required stack frames by eliminating no longer required frames. Due to no additional calculations on the returned value of the recursive call, it's safe to modify the current stack frame. There's no need to preserve previous stack frames because the control doesn't need to go back to the parent function. That reduces the stack frame space complexity of the recursive call from `O(N)` to `O(1)`, resulting in faster and more memory-friendly machine-code. Well, at least in languages supporting tail-call optimization/elimination.

PROJECT LOOM

Project Loom, an effort to support easy-to-use, high-throughput lightweight concurrency and new programming models, will add support for stack frame manipulation. The JVM gains support for unwinding the stack to some point and invoking a method with given arguments, a feature called *unwind-and-inkove*.

That allows for efficient tail-calls, but automatic tail-call optimization is not an explicitly stated project goal. Nevertheless, these are pleasant changes that might lower the barriers to use recursion if the necessary tools are available.

Streams to the Rescue

Java might not directly support tail-call optimization, but that doesn't mean you can't implement a better way to do recursive style coding yourself, with a little help of lambda expressions and Streams.

Thanks to the infinite nature of Streams, you can build a pipeline that runs until reaching a recursive base condition. But instead of calling the lambda expression recursively, it returns a new expression that runs in the Stream pipeline. This way, the stack depth will remain constant, regardless of the number of performed steps.

Our example of calculating a factorial will result in a number overflow[2] before a `StackOverflowError` occurs, so a simpler example is needed, like summing up consecutive numbers ($1 + 2 + \ldots + n$). The recursive variant is shown in Example 6-3, and it fails on my machine with a `StackOverflowError` for $n > 3571$.

Example 6-3. Recursivley summing up consecutive numbers

```
long sum(long total, long summand) {
  if (summand == 1L) {
    return total;
  }
```

```
  return sum(total + summand, summand - 1L);
}

var result = sum(1L, 4000L); // => StackOverflowError
```

To run the calculation in a Stream, you need to create a functional interface for wrapping the recursion, as seen in Example 6-4.

Example 6-4. Recursive-like Functional Interface

```
@FunctionalInterface
public interface RecursiveCall<T> {

  RecursiveCall<T> apply(); ❶

  default boolean isComplete() { ❷
    return false;
  }

  default T result() { ❸
    throw new Error("not implemented");
  }

  default T run() { ❹
    return Stream.iterate(this, RecursiveCall::apply)
                 .filter(RecursiveCall::isComplete)
                 .findFirst()
                 .get()
                 .result();
  }

  static <T> RecursiveCall<T> done(T value) { ❺

    return new RecursiveCall<T>() {

      @Override
      public boolean isComplete() {
        return true;
      }

      @Override
      public T result() {
        return value;
      }

      @Override
      public RecursiveCall<T> apply() {
        throw new UnsupportedOperationException();
```

```
      }
    };
  }
}
```

❶ The method `apply()` represents the recursive call. It executes the recursive step and returns a new lambda with the next step.

❷ The wrapper needs to know when it reaches a base condition, and the call-chain is complete.

❸ Because the lambda returns a new lambda instead of the result of its calculation, the wrapper needs a way to access the actual result.

❹ Calling `run()` will create and run an infinite Stream pipeline. The `Stream.iterate(…)` method applies the initial value (`this`) to an `UnaryOperator` (`this::apply`). The result is then iteratively applied again to the `UnaryOperator`. The applying of the result to the operator is done until the Stream's terminal operation is reached.

❺ A convenience method for creating a lambda representing a reached base condition and containing the actual result.

This simple functional interface is an iterative wrapper for recursive-style calls, eliminating the `StackOverflowError`. The previous recursive example doesn't differ much if you use the wrapper instead, as seen in Example 6-5.

Example 6-5. Summing up numbers recursively

```
RecursiveCall<Long> sum(Long total, Long summand) {
  if (summand == 1) {
    return RecursiveCall.done(total);
  }

  return () -> sum(total + summand, summand - 1L);
}

var result = sum(1L, 4000L).run();
```

Compared to the "real" recursive version, the only difference is that the base condition and the return value of `sum(…)` are wrapped in a new `RecursiveCall` lambda. Also, you must call `run()` to start the pipeline. See Figure 6-2 for how the Stream pipeline works iteratively on the recursive problem.

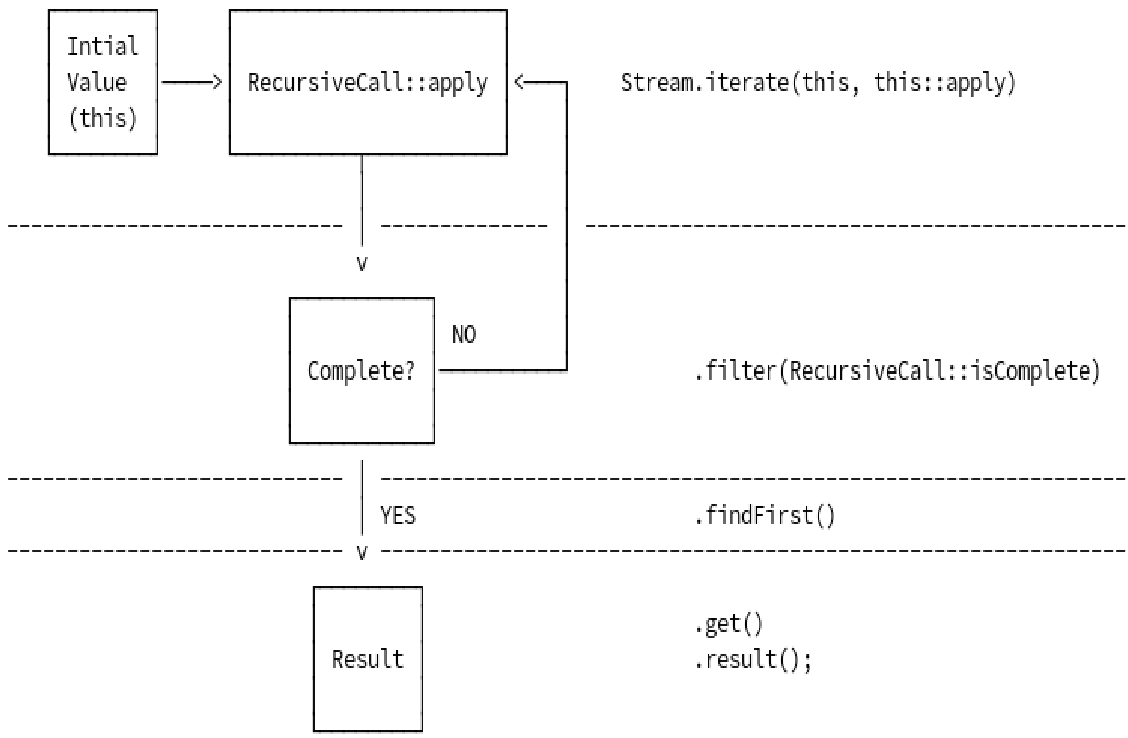

Figure 6-2. Stream-based recursion flow

The invisible difference between recursion and "recursion-like" with streams is the stack depth. As the recursive version creates a new stack frame for every method invocation, the Stream works iteratively, and therefore has a consistent stack depth[3]. That allows you to use a recursive approach to solving a problem, but without the possibility of a `StackOverflowError`.

A More Complex Example

So far, you've seen examples for factorials and summing up numbers. As good as they are for explaining recursion, they don't reflect "real-world" applications. That's why it's time to look at a more realistic example: traversing a tree-like data structure, as seen in Figure 6-3.

Recursive Tree-Traversal

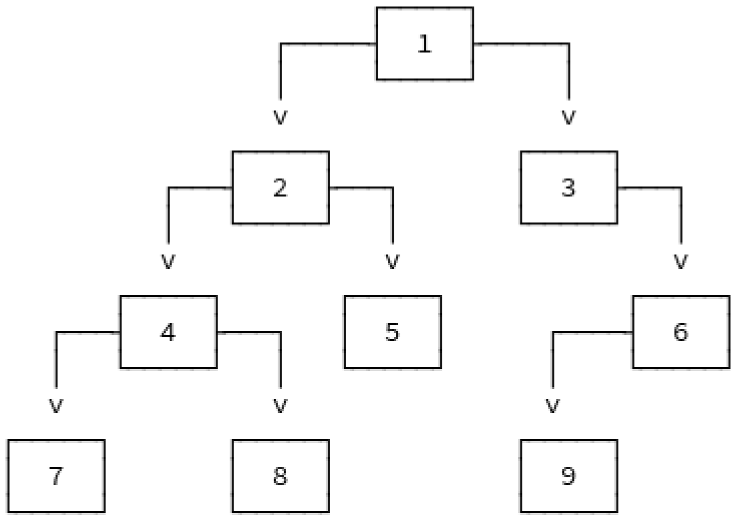

Figure 6-3. Tree-like data structure traversal

The data structure has a single root node, and every node has an optional left and right child node. There are multiple ways to traverse a tree. This example is "in-order", meaning it will traverse every node's left child node until no other node is reachable. Then it will continue to traverse down the right child's left nodes before going up again.

Thanks to records, as introduced in [Link to Come], a `Node<T>` is representable by the record definition itself. Adding the traversal method directly to `Node<T>` allows a node to traverse its children, regardless of the tree's actual root node. The `Node<T>` definition can be see in Example 6-6.

Example 6-6. Tree node and recursion

```
public record Node<T>(T value,
                      Node<T> left,
                      Node<T> right) {

  private <T> void traverse(Node<T> node) { ❶
    if (node == null) { ❷
      return;
    }

    traverse(node.left); ❸
```

```
    System.out.print(node.value + " "); ❹

    traverse(node.right); ❺
  }

  public <T> void traverse() { ❻
    traverse(this);
  }
}

// Building the tree-like data structure
var node7 = new Node<String>("7", null, null);
var node8 = new Node<String>("8", null, null);
var node4 = new Node<String>("4", node7, node8);

var node5 = new Node<String>("5", null, null);
var node2 = new Node<String>("2", node4, node5);

var node9 = new Node<String>("9", null, null);
var node6 = new Node<String>("6", node9, null);
var node3 = new Node<String>("3", null, node6);

var node1 = new Node<String>("1", node2, node3);
```

❶ The traversal method itself is `private`, because nodes should only traverse themselves and their children.

❷ The base condition

❸ Traverse the left child.

❹ Print the current node value.

❺ Traverse the right child. That will lead to traversing a possible left grandchild next.

❻ The `public` traverse method to start the recursive call-chain.

The tree is traversed by invoking `node1.traverse()`, which outputs the expected sequence: *7 4 8 2 5 1 3 6 9*.

The code is concise and easy to understand. Let’s look at an iterative approach for comparison.

Iterative Tree-Traversal

Compared to the recursive approach, traversing a tree iteratively needs more lines of code and is more complicated, as seen in Example 6-7.

Example 6-7. Iterative tree traversal

```
var nodeStack = new Stack<Node<String>>(); ❶
var current = node1;❶

while(!nodeStack.isEmpty() || current != null) { ❷

  if (current != null) { ❸
    nodeStack.push(current);
    current = current.left();
    continue;
  }

  current = nodeStack.pop(); ❹

  System.out.println(current.value() + " "); ❺

  current = current.right(); ❻
}
```

❶ Auxiliary variables are needed to save the current state of the iteration.

❷ Iterate as long as a node is present, or `nodeStack` isn't empty.

❸ A `java.util.Stack` saves all nodes until the bottom is reached.

❹ At this point, the loop can't go deeper because it encountered `current == null`, so it sets `current` to the last node saved in `nodeStack`.

❺ Output the node value, just like before in the recursive version.

❻ Traverse the right child.

The output is the same as before: *7 4 8 2 5 1 3 6 9*.

But the code doesn't have the conciseness of the recursive approach. Two additional variables are needed, and the general logic is more convoluted.

When (Not) To Use Recursion

Recursion is an often overlooked technique. It's easy to get it wrong (non-working base-case), can be harder to understand (especially if you're not used to it), and has an unavoidable overhead resulting in slower execution times than iterative structures.

You should always consider the additional overhead and stack-overflow problems when choosing between recursion and its alternatives. If you're running in a JVM with ample memory available and a big enough stack size, even bigger recursive call-chains won't be a problem. But if your problem size is unknown or not fixed, an alternative approach can prevent a `StackOverflowError` in the long run.

Some scenarios are better suited for a recursive approach, though, even in Java without tail-call optimization. Especially if you're dealing with self-referencing data structures like linked lists or trees, recursion will feel more natural. Traversing tree-like structures can also be done iteratively but will most likely result in more complex code that's harder to reason with. And that will hurt long-time maintainability.

Which to choose — recursion or iteration — depends highly on the problem you want to solve and in which environment your code runs. Recursion is often the preferred tool for solving more abstract problems, and iteration is preferred for more low-level code. Iteration might provide better runtime performance, but recursion can improve your productivity as a programmer.

Table 6-1. Recursion versus iteration

	Recursion	Iteration

Implementation	Self-calling function	Loop
State	Stored on Stack	control variables (.e.g. a loop index)
Progress	Towards base condition	Towards control value condition
Termination	Base condition reached	Control variable condition reached
Verbosity	↘	↗
If not terminated	`StackOverflowError`	endless loop
Overhead	↗	↘
Performance	↘	↗

Takeaways

- Recursion is the functional alternative to traditional iteration.
- It's best used for partially solvable problems.
- Java lacks tail-call-optimization, which can lead to `StackOverflowExceptions`.
- Streams can provide an alternative approach.
- Don't force recursion for functional's sake. Use what fits the context best.

---

1 The default stack size of the most JVMs is 1MB. You can set a bigger stack size with the flag `-Xss`. Please see the Oracle Java Tools Documentation for more information.

2 The biggest possible `long` is 9,223,372,036,854,775,807, or 2^63-1. This value lies between 20! and 21!. The number overflows way before the stack overflows.

3 The actual stack depth depends on the underlying Stream. It might differ if its implementation changes. But it will always be consistent, regardless of the required recursive steps.

Chapter 7. Exception Handling

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

Exceptions are Java's mechanism of choice for handling *disruptive* and *abnormal* control flow conditions of your programs. The general concept of exceptions traces back to the origins of Lisp[1] and is used by many different programming languages[2]

Java' exception handling mechanisms slightly improved over time, like adding support for catching multiple types of exceptions at once with a single `try-catch`-block (`multi-catch`) or better handling of resources (`try-with-resources`). But so far, no improvements targeted at lambdas have found their way into the JDK.

This chapter will show you the different kinds of exceptions and their impact on functional programming with lambdas. You will learn how to handle exceptions in lambdas or alternative ways to approach control flow disruptions in a functional context.

Java Exception Handling in a Nutshell

There are three different kinds of control flow disruptions in Java, with disparate requirements regarding their handling in your code: *checked* and *unchecked* exceptions, and *errors*.

Checked exceptions are (supposed to be) *anticipated* and *recoverable* events outside of normal control flow. You should always expect the possibility of a missing file (`java.io.FileNotFoundException`) or an invalid URL (`java.net.MalformedURLException`). And because they're anticipated, they must adhere to the *catch-or-specify* requirement.

> **CATCH-OR-SPECIFY**
>
> The *catch-or-specify* requirement declares that your code must honor one of the following conditions while dealing with *checked* exceptions:
>
> - *Catch*: An appropriate handler — a `catch`-block — for the exception, or one of its base types, is provided.
> - *Specify*: The surrounding method signifies its thrown exception types by using the `throws` keyword, followed by a comma-separated list of possible *checked* exceptions.
>
> This requirement **must** be obliged, and the compiler will make sure that you fulfill at least one of them. There's no need to specify an exception type if you catch and handle it. An unnecessary `throws` forces the consumer of such a method to comply with the *catch-or-specify* requirement, too.

This requirement intends to locally flag possible exceptional states or force you to handle it directly. That leads to improved software reliability and resilience by giving you the possibility to recover gracefully or hand over the liability down the line instead of ignoring the exception completely.

Unchecked exceptions, on the other hand, are *not anticipated*, and are often *unrecoverable*, like:

- Unsupported operations: `java.lang.UnsupportedOperationException`
- Invalid mathematical calculations: `java.lang.ArithmeticException`
- Empty references: `java.lang.NullPointerException`

They aren't considered part of the methods' contract but rather represent what happens if the contract is broken. Therefore, such exceptions aren't subject to *catch-or-specify*, and methods usually don't signify them with the `throws` keyword, even if it's known that a method will throw them. They still have to be handled in some form, though, if you don't want your program to crash. If not handled directly, an exception automatically goes up the call stack of the current thread until it finds an appropriate handler. Or, if none is available, the thread dies. For single-threaded applications, this means the runtime will terminate, and your program crashes.

The third kind of control flow disruptions — *errors* — indicate a severe problem you shouldn't catch or can't handle under normal circumstances. For example, if the runtime runs out of available memory, it throws a `java.lang.OutOfMemoryError`. Or an endless recursive call will eventually lead to a `java.lang.StackOverflowError`. There's nothing you could really do without any memory left, regardless of whether it's the heap or the stack. Faulty hardware is another source for Java *errors*, like `java.io.IOError` in case of a disk error. These are all grave problems with almost no possibility to recover gracefully. Because they also aren't anticipated, they mustn't adhere to *catch-or-specify*.

All exceptions are *checked*, except types subclassing `java.lang.RuntimeException` or `java.lang.Error`. But they share a common base type: `java.lang.Throwable`. Types inheriting from the latter two are either *unchecked*, or an *Error*. You can see the hierarchy in Figure 7-1.

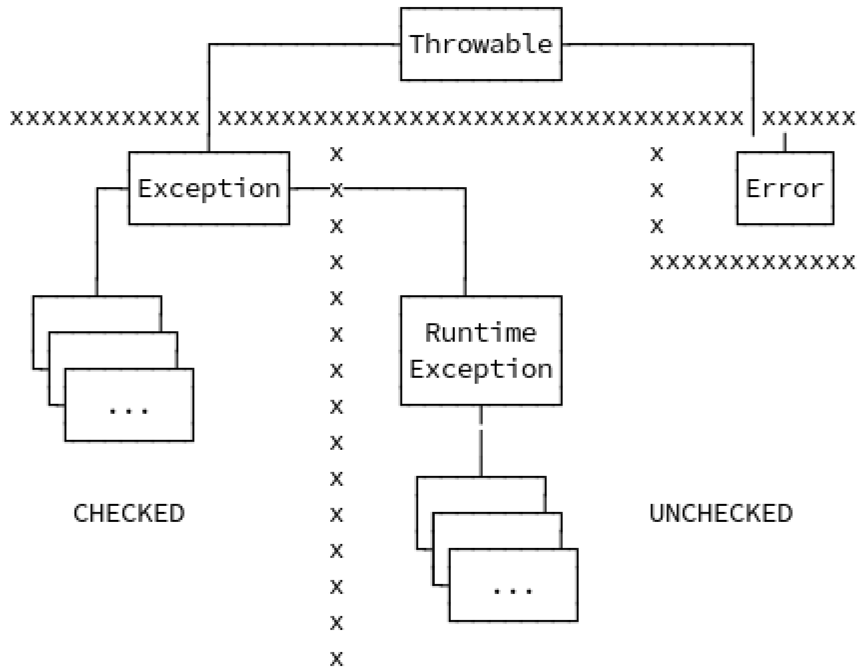

Figure 7-1. Exceptions hierarchy in Java

Checked Exceptions and Lambdas

Java's exception handling existed since its inception. It was designed to fulfill specific requirements at the time, independent from any possible requirements that might arise 18 years later with with the introduction of lambdas. That's why throwing and handling exceptions don't fit nicely into a functional Java coding style without any special considerations.

Even the simplest example, like Example 7-1, is quite verbose and can lead to functional impurity by making the code no longer deterministic. This particular example is just a placeholder for the underlying problem. But it's transferrable to any code dealing with checked exceptions in lambdas.

Example 7-1. Checked Exceptions in Lambdas

```
String read(File input) throws IOException { ❶
  // ...
}

Stream<File> files = Stream.of(...);
```

```
// COMPILER ERROR: Unhandled Exception of type IOException

files.map(this::read) ❷
      .filter(Objects::nonNull)
      .map(String::toUpperCase)
      .forEach(System.out::println);


// YOU NEED TO USE TRY-CATCH

files.map(file -> {
  try { ❸
    return read(file);
  }
  catch (IOException e) {
    // handle the exception...
    return null;
  }
}).filter(Objects::nonNull)
  .map(String::toUpperCase)
  .forEach(System.out::println);
```

❶ The method signature indicates a checked exception, so any code calling it must adhere to the *catch or specify requirement*.

❷ Calling `read(File file)` as a method reference is the most concise way to use it in the Stream pipeline and should be preferred. But thanks to the checked `IOException`, it can't be used that way.

❸ The exception needs to be handled locally, introducing a `try`-`catch`-block into the Stream operation.

As you can see, using a `try`-`catch`-block directly in lambdas is a cumbersome and quite ugly way to deal with exceptions. But there are different approaches to handling exceptions in lambdas without losing (most) of the simplicity and clarity that lambdas, methods references, and Streams provide.

Safe Method Extraction

How to handle exceptions in your Streams highly depends on who controls the code. If the throwing code is entirely under your control, you should *always* adequately handle its exceptions. But often, the offending code is *not* yours, or you can't change or refactor it as needed. In this case, you can still extract it into a "safer" method with appropriate local exception handling, as shown in Example 7-2.

Example 7-2. Safe method wrapper

```
String read(File input) throws IOException { ❶
  // ...
}

String safeRead(File file) { ❷
  try {
    String content = read(file);
  }
  catch (IOException e) {
    // ...
    return null; ❸
  }
}

Stream<File> files = ...

files.map(this::safeRead) ❹
     .filter(Objects::nonNull)
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

❶ The throwing method that might not be under your control, or you don't want or can't refactor the original behavior.

❷ A "safe" wrapper method is introduced to handle the exception instead.

❸ Handling the error locally and return an adequate fallback value.

❹ The wrapper method allows the use of a method reference, making the code concise and readable again.

Creating a "safe" method allows you to handle the exception locally in any way necessary in its original context. The pipeline remains robust if the exception is handled gracefully. You also have a chance for additional actions, like logging But how you handle the *checked* exception is up to you and depends on your requirements. This isn't a "one-size-fits-all" approach because sometimes there's no way to handle the exception locally.

> **NOTE**
>
> This approach can be seen as a more localized version of the *facade pattern* [3]. Instead of wrapping a whole class to provide a safer, context-specific interface, only specific methods get a new facade to improve their handling for certain use-cases. That allows you to reduce the affected code and still gain the advantages of a facade, like reduced complexity and improved readability. Also, it's a good starting point for future refactoring efforts.

Safe wrapper methods are an improvement over using `try-catch` blocks in a lambda because you keep the expressiveness of inline-lambdas and method references and have a chance to handle any exceptions. But it's still only another abstraction over existing code to regain control of disruptive conditions.

Not Throwing Exceptions in the First Place

Finding a better way to handle exceptions, especially in lambdas, is a worthwhile endeavor. But if you have control over the API, you could design its contracts to make exceptions unnecessary instead. Or at least more manageable.

As seen in Chapter 5, Java has a specialized type representing the absence of values: `Optional<T>`. With its help, you can refrain from returning `null` as much as possible to mitigate the dreaded `NullPointerException` gracefully. But keep in mind that `null` isn't always equivalent to the absence of a value. Returning `null` can have a completely different meaning! It highly depends on your requirements and how you designed your API contracts in the first place.

Let's look at the previous example with Optionals, by modifying the `read` method, as seen in Example 7-3.

Example 7-3. Optional versus Exception

```
Optional<String> read(File input) { ❶
  try {
    // ...
    return Optional.ofNullable(...);
```

```
  }
  catch (IOException e) {
    // ...
    return Optional.empty(); ❷
  }
}

Stream<File> files = Stream.of(...);

files.map(this::read)
     .flatMap(Optional::stream) ❸
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

❶ The previously throwing method changed its signature to return an `Optional<String>` instead.

❷ In case of an exception, an empty Optional is returned. Therefore, no additional “safe” method is required.

❸ The Stream can unpack the Optional directly with `flatMap`.

At first look, we didn’t gain much compared to a “safe” method handling the `IOException` and returning `null`. Like with “safe” methods, this approach is highly dependent on your requirements. Any exception handling is still localized to its occurrence and can’t be delegated. But by changing the `read` method itself to an `Optional<String>`, no additional “safe” method is required, and the returned value provides the functional and fluent API of an Optional.

But in the end, it’s just another mere “band-aid” to ease the pain, but not fixing any of the foundational problems of exceptions and lambdas.

The Anti-Pattern: Unchecking Exceptions

There’s another and most consequential way of “not dealing with exceptions” that destroys the fundamental idea behind checked exceptions for the sake of making the compiler happy: *unchecking exception*.

Instead of dealing with a *checked* exception directly, you *hide* it in an *unchecked* exception. This is usually done by creating specialized `@FunctionalInterface` to wrap the offending lambda or method reference. It catches the original exception and rethrows it as an *unchecked* `RuntimeException`, or one of its siblings.

Congratulations, the compiler is happy now and won't force you to handle the exception anymore. But the wrapper type doesn't fix the original problem of possible control flow disruption. Instead, there's no exception handling at all. It's just hidden away, sweeping the problem "under the carpet" by circumventing *catch-or-specify*. And any exception still disrupts the control flow without being appropriately handled.

Unchecked exceptions are supposed to be unanticipated and are often unrecoverable. That's why they don't fall under the *catch-or-specify requirement* in the first place! It abuses the different kinds of exceptions and the requirements for their handling by Java.

A More Functional Approach to Exceptions?

Exceptions are supposed to be additional signals about your control flow. APIs can and should use exceptions purposefully. Before Java 8 introduced lambdas, Streams, and method references, exceptions would fit nicely in the available constructs and the mainly imperative coding style. In the last few years, though, Java is finally advancing faster than ever before. But not all of its parts keep up with the general pace. You have to find a reasonable compromise between a functional approach to your code and more *traditional* constructs for managing control flow.

You have to remember that Java is a *general-purpose* language with class-based object orientation at its core. Its exception handling clearly shows its primarily imperative coding style. Even with all the functional additions since version 8, it didn't become a full-fledged functional language overnight. But we can look for inspiration in another, more functional language again: *Scala*.

Try/Success/Failure in Scala

Scala is arguably the closest functional relative to Java available on the JVM, not considering Clojure for its more foreign syntax. It addresses many of Java's "shortcomings," is functional at its core, and has an excellent way of dealing with exceptional conditions.

The *Try/Success/Failure* pattern and its related types `Try[+T]`, and its derived types `Success[+T]` and `Failure[+T]` [4], are Scala's way of dealing with exceptions in a more functional fashion. You can think of it as specialized Java `Optional<T>` with integrated exception handling. Where an `Optional<T>` indicates that a value might be missing, `Try[+T]` can tell you *why*. And instead of being *just* a generic wrapper around another object, Scala supports *pattern-matching*, a `switch`-like concept of handling the different outcomes. That allows for quite concise and straightforward exception handling.

A `Try[+T]` can either result in a `Success[+T]` or `Failure[+T]`, with the latter containing a `Throwable`. Even without full knowledge of Scala syntax, the code in Example 7-4 should be clear.

Example 7-4. Scala Try/Success/Failure

```
def read(file: File): Try[String] = Try { ❶
  ... // code that will throw an exception
}

val file = new File(...);

result read(file) { ❷
  case Success(value) => println(value.toUpperCase) ❸
  case Failure(e) => println("Couldn't read file: " + e.getMessage)
❹
}
```

❶ The return type is an `Try[String]`, so the method must either return a `Success[String]` containing the content of the `File`, or a `Failure[Throwable]`. Scala doesn't need an explicit `return` and returns the last value. Any exception is caught by the `Try { ... }` construct.

Pattern matching simplifies the result handling. The cases are lambdas

❷ Pattern matching simplifies the result handling. The cases are lambdas, and the whole block is similar to an Optional call-chain with `map(…)` and `orElse(…)`.

❸ `Success` provides access to the return value, like `Some`.

❹ If an exception occurs, you handle it with the `Failure` case.

`Try[+A]` is a great Scala feature, combining the concept of Optionals and exception handling into a single, easy-to-use type. But what does that mean for you as a Java developer? Let's try to implement something similar with Java ourselves!

Try/Success/Failure in Java

Java doesn't provide anything out-of-the-box that comes close to the *Try/Success/Failure* pattern. But the general concept can be implemented in Java, although it lacks the conciseness and elegance of the Scala version and its pattern matching.

A minimalistic implementation like in Example 7-5 requires less than 50 lines of code.

Example 7-5. Java Try/Success/Failure

```
import java.util.Objects;
import java.util.function.Consumer;
import java.util.function.Function;

public class Try<T, R> { ❶

    private Function<T, R> fn;

    private Function<RuntimeException, R> failureFn;

    public static <T, R> Try<T, R> of(Function<T, R> fn) { ❷
        Objects.requireNonNull(fn);
        return new Try<>(fn, null);
    }

    private Try(Function<T, R> fn,
                Function<RuntimeException, R> failureFn) { ❷
        this.fn = fn;
        this.failureFn = failureFn;
```

```
    }

    public Try<T, R> success(Function<R, R> successFn) { ❸
        var composedFn = this.fn.andThen(fnOut -> {
            successFn.apply(fnOut);
            return fnOut;
        });

        this.fn = composedFn;

        return this;
    }

    public Try<T, R> failure(Function<RuntimeException, R>
failureFn) { ❹
        Objects.requireNonNull(failureFn);
        this.failureFn = failureFn;

        return this;
    }

    public Optional<R> apply(T value) { ❺
        try {
            return Optional.ofNullable(this.fn.apply(value));
        }
        catch (RuntimeException e) {
            if (this.failureFn != null) {
                var failureResult = this.failureFn.accept(e);
                return Optional.ofNullable(failureResult);
            }
        }

        return Optional.empty();
    }
}

// WON'T COMPILE!

Optional<String> maybeContent = Try.<File, String> of(this::read)
//
                                   .success(String::toUpperCase) //
                                   .failure(e -> ...)) //
                                   .apply(new File(...));
```

❶ This particular `Try` implementation wraps `Function<T, R>`, so it must match the generic signature.

❷

The convenience `static of(…)` method simplfies the creation of `Try` objects. The `private` constructor disallows direct instantiation, similar to `Optional<T>`.

❸ The `success` case provides a `Function<T, T>` by functionally composing the original function with it. This allows you to work on the successful result of the initial function before returning it.

❹ If an exception occurs, you handle it in `failure` case.

❺ The `apply(T value)` method is realizing the lazy `Try` call-chain. In case of an exception but no handler, the exception is swallowed whole.

Even though this naïve implementation lacks flexibility, you can clearly see its intention to handle a certain workflow more functionally. With the result being an `Optional<T>`, you can extend the call-chain even further. But it still suffers from the two main issues making such implementations a chore: First, it only supports methods references or lambdas with unchecked exceptions as an input. You can't get around the *catch-and-specify* requirement without "unchecking" the original exception, which is highly discouraged. And second, it only supports a single functional interface and its equivalents. You would have to implement `Try` repeatedly for different functional interface signatures.

In this particular use-case, the previously mentioned *anti-pattern* of unchecking exception can be helpful to mitigate and allows you to handle any exception. You need to create our own `Function<T, R>` derivate to accept checked exceptions, as seen in Example 7-6.

Example 7-6. Function<T, R> with Checked Exceptions

```
@FunctionalInterface
public interface CheckedFunction<T, R> {

  R apply(T t) throws Exception;
}
```

The new type lacks most functionality that `Function<T, R>` provides, but it allows checked exceptions. `Try<T, R>` needs to be adapted, too. But it's almost a *drop-in* replacement as shown in Example 7-7.

Example 7-7. Try<T, R> with Checked Exceptions

```
import java.util.Objects;
import java.util.Optional;
import java.util.function.Function;

public class Try<T, R> {

  private CheckedFunction<T, R> fn; ❶

  private Function<Exception, R> failureFn;

  public static <T, R, Exception> Try<T, R> of(CheckedFunction<T,
R> fn) {
      Objects.requireNonNull(fn);
      return new Try<>(fn, null);
    }

    private Try(CheckedFunction<T, R> fn, Function<Exception, R>
failureFn) {
        this.fn = fn;
        this.failureFn = failureFn;
    }

    public Try<T, R, E> success(CheckedFunction<R, R> successFn) {
❷
        var prev = this.fn;
        this.fn = in -> successFn.apply(prev.apply(in));
        return this;
    }

    public Try<T, R> failure(Function<Exception, R> failureFn) {
        Objects.requireNonNull(failureFn);
        this.failureFn = failureFn;

        return this;
    }

    public Optional<R> apply(T value) {
        try {
            return Optional.ofNullable(this.fn.apply(value));
        }
        catch (Exception ex) {
            if (this.failureFn != null) {
                var failureResult = this.failureFn.apply(ex);
                return Optional.ofNullable(failureResult);
            }
        }
```

```
        return Optional.empty();
    }
}

// IT FINALLY COMPILES!

Optional<String> maybeContent = Try.<File, String> of(this::read)
//
                                  .success(String::toUpperCase) //
                                  .failure(e -> ...)) //
                                  .apply(new File(...));
```

❶ Instead of `Function<T, R>` the new type `CheckedException<T, R>` is used.

❷ The new type doesn't support functional composition with `andThen(…)`, but the actual code for doing so is trivial.

The code finally compiles!

But without providing a lot of additional unchecking functional interfaces, it's not a practical solution. There are many third-party libraries available that have already done most of the work for you. Two such functional libraries are the Vavr project and jOOλ, providing more flexible tools in the vein of our `Try` type.

Functional Exceptions with CompletableFuture

In [Link to Come] you've learned about `CompletableFuture<T>`, an already available fluent API for doing work with lambdas and handling exceptions. On the surface, it's quite identical to the custom `Try` implementation, as seen in Example 7-8. But it still shares the same fundamental kryptonite: *checked* exceptions.

Example 7-8. Function Exceptions with CompletableFuture

```
String content = CompletableFuture.supplyAsync(() -> read(file))
                                  .exceptionally(ex -> null))
                                  .thenApply(String::toUpperCase)
                                  .get();
```

Even if we ignore its inability to handle *checked* exceptions in a concise way, it's still not the perfect tool for functional exception handling, thanks to its reliance on threads. `CoompletableFuture` provides a simple interface to interconnect multiple steps that run asynchronously and trigger their respective parts on completion. So the introduced overhead has to be considered, making it a bad match for synchronous or simple problems.

How to Choose Your Approach

Exception handling can be quite a pain point in Java, regardless of a functional approach. There is always a trade-off, no matter which presented option you choose, especially if checked exceptions are involved.

- Extracting unsafe methods to gain localized exception handling is a better compromise but not an easy-to-use general solution.
- Designing your APIs to not use exceptions at all is not as easy as it sounds.
- Unchecking your exceptions is a "last-resort" tool that hides them away without a chance to handle them and contradicts their purpose.

So what should you do? Well, it depends.

None of the presented solutions is *perfect*. You have to find a balance between "convenience" and "usability." Exceptions are sometimes an overused feature, but they are still essential signals to the control flow of your programs. Hiding them away might not be in your best interest in the long run, even if the resulting code is more concise and reasonable, as long as no exception occurs.

Not every imperative or OOP feature/technique is replaceable with a functional equivalent in Java. Many of Java's (functional) shortcomings are circumventable to gain their general advantages, even if the resulting code is not as concise as in fully-functional programming languages. But exceptions are one of those features that aren't easily replaceable in most circumstances. They're often an indicator that you either need to refactor your code to make it “more functional” or that a functional approach might not be the best solution for the problem.

Alternatively, there are several third-party libraries available, like the Vavr project or jOOλ, that allow you to circumvent the general problems with using (checked) exceptions in functional Java code. They did all the work implementing all relevant wrapper interfaces and replicating control structures and types from other languages, like pattern matching. But in the end, you end up with highly specialized code that tries to bend Java to its will, without much regard for traditional or common code constructs. Such a dependency is a long-term commitment and shouldn’t be added lightly.

Takeaways

- There’s no specialized exception handling for lambdas, only `try-catch` as usual, which leads to verbose and unwieldy code.
- You can fulfill or circumvent the *catch-or-specify* in multiple ways, but that merely hides the original “problem.”
- Custom wrappers can provide a more functional approach.
- Third-party libraries can help to reduce the additional boilerplate required for handling exceptions more functionally. But the newly introduced types and constructs are no lightweight addition to your code and might create a lot of technical debt.
- No general approach is available. Choosing the right way to handle exceptions depends highly on the surrounding context.
- Often, an imperative approach is more recommended than trying to work around the limitations of lambdas regarding exceptions.

---

1 Guy L. Steele and Richard P. Gabriel. 1996. “The evolution of Lisp.” History of programming languages---II. Association for Computing Machinery, 233-330.

2 The Wikipedia entry on Exception handling syntax provides an overview of different kinds of syntaxes and languages.

3 Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Boston, MA: Addison Wesley.

4 Scala's generic types are declared with `[]` (sqare brackets) instead of <> (angle brackets). The + (plus) signifies the type's variance. See "Tour of Scala" for more information about type variance.

About the Author

Using his first computer at the age of four, **Ben Weidig** is a self-taught developer with almost two decades of experience in professional web, mobile, and systems programming in various languages.

After learning the ropes of professional software development and project management at an international clinical research organization, he became a self-employed software developer. He merged with a SaaS company after prolonged and close collaboration on multiple projects. As co-director, he shapes the company's general direction, is involved in all aspects of their Java-based main product, and oversees and implements its mobile strategy.

In his free time, he shares his expertise and experiences by writing articles about Java, functional programming, best practices, and code-style in general. He also participates in Open-Source, either as a committer to established projects or releasing code of his own.

Colophon

The animal on the cover of *FILL IN TITLE* is *FILL IN DESCRIPTION*.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *FILL IN CREDITS*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.