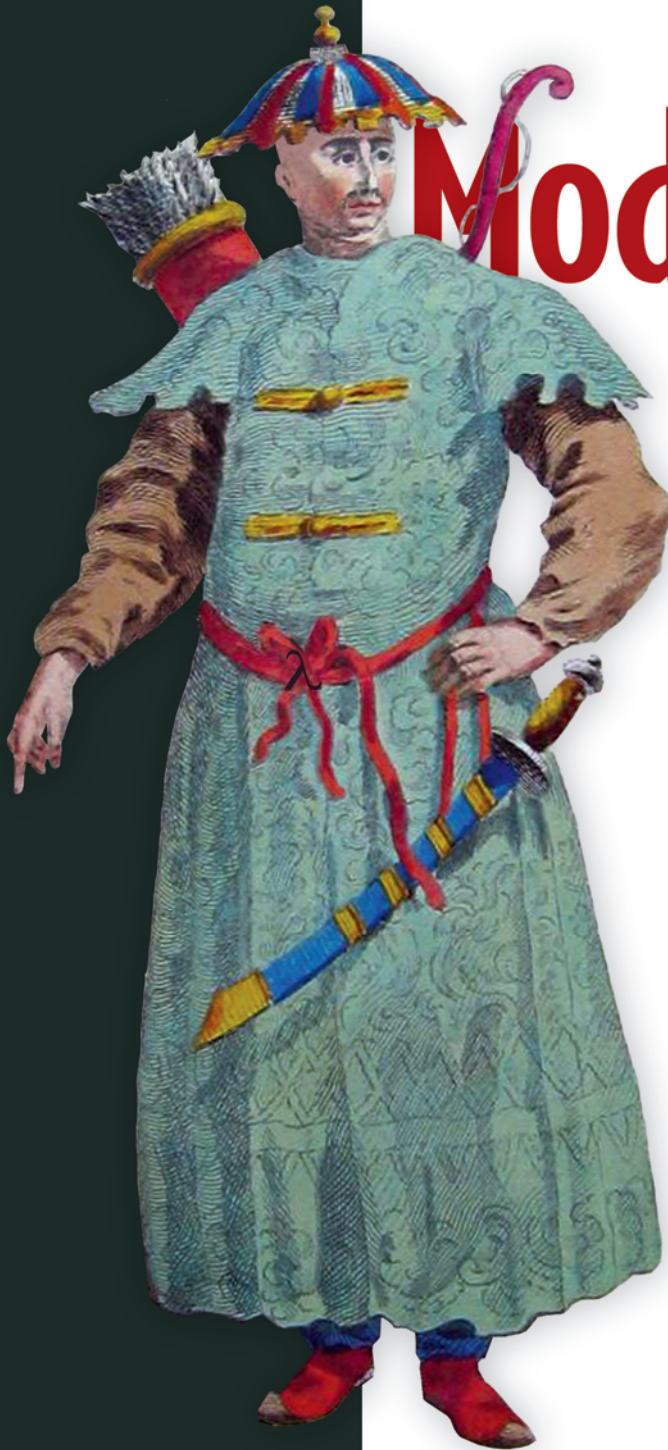


Lambdas, streams, functional and reactive programming



# Modern Java IN ACTION

Raoul-Gabriel Urma  
Mario Fusco  
Alan Mycroft



## *Modern Java in Action*

by Raoul-Gabriel Urma, Mario Fusco,  
and Alan Mycroft

### **Chapter 1**

Copyright 2018 Manning Publications

# *brief contents*

---

<b>PART 1</b>	<b>FUNDAMENTALS .....</b>	<b>1</b>
1	■ Java 8, 9, 10, and 11: what's happening?	3
2	■ Passing code with behavior parameterization	26
3	■ Lambda expressions	42
<b>PART 2</b>	<b>FUNCTIONAL-STYLE DATA PROCESSING WITH STREAMS ...</b>	<b>79</b>
4	■ Introducing streams	81
5	■ Working with streams	98
6	■ Collecting data with streams	134
7	■ Parallel data processing and performance	172
<b>PART 3</b>	<b>EFFECTIVE PROGRAMMING WITH STREAMS AND LAMBDA.....</b>	<b>199</b>
8	■ Collection API enhancements	201
9	■ Refactoring, testing, and debugging	216
10	■ Domain-specific languages using lambdas	239

<b>PART 4</b>	<b>EVERYDAY JAVA .....</b>	<b>273</b>
11	■ Using Optional as a better alternative to null	275
12	■ New Date and Time API	297
13	■ Default methods	314
14	■ The Java Module System	333
<b>PART 5</b>	<b>ENHANCED JAVA CONCURRENCY .....</b>	<b>355</b>
15	■ Concepts behind CompletableFuture and reactive programming	357
16	■ CompletableFuture: composable asynchronous programming	387
17	■ Reactive programming	416
<b>PART 6</b>	<b>FUNCTIONAL PROGRAMMING AND FUTURE JAVA EVOLUTION .....</b>	<b>443</b>
18	■ Thinking functionally	445
19	■ Functional programming techniques	460
20	■ Blending OOP and FP: Comparing Java and Scala	485
21	■ Conclusions and where next for Java	500

# 1

## *Java 8, 9, 10, and 11: what's happening?*

---

### ***This chapter covers***

- Why Java keeps changing
- Changing computing background
- Pressures for Java to evolve
- Introducing new core features of Java 8 and 9

Since the release of Java Development Kit (JDK 1.0) in 1996, Java has won a large following of students, project managers, and programmers who are active users. It's an expressive language and continues to be used for projects both large and small. Its evolution (via the addition of new features) from Java 1.1 (1997) to Java 7 (2011) has been well managed. Java 8 was released in March 2014, Java 9 in September 2017, Java 10 in March 2018, and Java 11 planned for September 2018. The question is this: Why should you care about these changes?

### **1.1    *So, what's the big story?***


We argue that the changes to Java 8 were in many ways more profound than any other changes to Java in its history (Java 9 adds important, but less-profound, productivity changes, as you'll see later in this chapter, while Java 10 makes much

smaller adjustments to type inference). The good news is that the changes enable you to write programs more easily. For example, instead of writing verbose code (to sort a list of apples in inventory based on their weight) like

```
Collections.sort(inventory, new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

in Java 8 you can write more concise code that reads a lot closer to the problem statement, like the following:

```
inventory.sort(comparing(Apple::getWeight));
```



**The first Java 8 code  
of the book!**

It reads “sort inventory comparing apple weight.” Don’t worry about this code for now. This book will explain what it does and how you can write similar code.

There’s also a hardware influence: commodity CPUs have become multicore—the processor in your laptop or desktop machine probably contains four or more CPU cores. But the vast majority of existing Java programs use only one of these cores and leave the other three idle (or spend a small fraction of their processing power running part of the operating system or a virus checker).

Prior to Java 8, experts might tell you that you have to use threads to use these cores. The problem is that working with threads is difficult and error-prone. Java has followed an evolutionary path of continually trying to make concurrency easier and less error-prone. Java 1.0 had threads and locks and even a memory model—the best practice at the time—but these primitives proved too difficult to use reliably in non-specialist project teams. Java 5 added industrial-strength building blocks like thread pools and concurrent collections. Java 7 added the fork/join framework, making parallelism more practical but still difficult. Java 8 gave us a new, simpler way of thinking about parallelism. But you still have to follow some rules, which you’ll learn in this book.

As you’ll see later in this book, Java 9 adds a further structuring method for concurrency—reactive programming. Although this has more-specialist use, it standardizes a means of exploiting the RxJava and Akka reactive streams toolkits that are becoming popular for highly concurrent systems.

From the previous two desiderata (more concise code and simpler use of multi-core processors) springs the whole consistent edifice captured by Java 8. We start by giving you a quick taste of these ideas (hopefully enough to intrigue you, but short enough to summarize them):

- The Streams API
- Techniques for passing code to methods
- Default methods in interfaces

Java 8 provides a new API (called Streams) that supports many parallel operations to process data and resembles the way you might think in database query languages—you express what you want in a higher-level manner, and the implementation (here the Streams library) chooses the best low-level execution mechanism. As a result, it avoids the need for you to write code that uses *synchronized*, which is not only highly error-prone but also more expensive than you may realize on multicore CPUs.<sup>1</sup>

From a slightly revisionist viewpoint, the addition of Streams in Java 8 can be seen as a direct cause of the two other additions to Java 8: *concise techniques to pass code to methods* (method references, lambdas) and *default methods* in interfaces.

But thinking of passing code to methods as a mere consequence of Streams downplays its range of uses within Java 8. It gives you a new concise way to express *behavior parameterization*. Suppose you want to write two methods that differ in only a few lines of code. You can now simply pass the code of the parts that differ as an argument (this programming technique is shorter, clearer, and less error-prone than the common tendency to use copy and paste). Experts will here note that behavior parameterization could, prior to Java 8, be encoded using anonymous classes—but we'll let the example at the beginning of this chapter, which shows increased code conciseness with Java 8, speak for itself in terms of clarity.

The Java 8 feature of passing code to methods (and being able to return it and incorporate it into data structures) also provides access to a range of additional techniques that are commonly referred to as *functional-style programming*. In a nutshell, such code, called *functions* in the functional programming community, can be passed around and combined in a way to produce powerful programming idioms that you'll see in Java guise throughout this book.

The meat of this chapter begins with a high-level discussion on why languages evolve, continues with sections on the core features of Java 8, and then introduces the ideas of functional-style programming that the new features simplify using and that new computer architectures favor. In essence, section 1.2 discusses the evolution process and the concepts, which Java was previously lacking, to exploit multicore parallelism in an easy way. Section 1.3 explains why passing code to methods in Java 8 is such a powerful new programming idiom, and section 1.4 does the same for Streams—the new Java 8 way of representing sequenced data and indicating whether these can be processed in parallel. Section 1.5 explains how the new Java 8 feature of default methods enables interfaces and their libraries to evolve with less fuss and less recompilation; it also explains the *modules* addition to Java 9, which enables components of large Java systems to be specified more clearly than “just a JAR file of packages.” Finally, section 1.6 looks ahead at the ideas of functional-style programming in Java and other languages sharing the JVM. In summary, this chapter introduces ideas that are successively elaborated in the rest of the book. Enjoy the ride!

---

<sup>1</sup> Multicore CPUs have separate caches (fast memory) attached to each processor core. Locking requires these to be synchronized, requiring relatively slow cache-coherency-protocol inter-core communication.

## 1.2 Why is Java still changing?

With the 1960s came the quest for the perfect programming language. Peter Landin, a famous computer scientist of his day, noted in 1966 in a landmark article<sup>2</sup> that there had *already* been 700 programming languages and speculated on what the next 700 would be like—including arguments for functional-style programming similar to that in Java 8.

Many thousands of programming languages later, academics have concluded that programming languages behave like ecosystems: new languages appear, and old languages are supplanted unless they evolve. We all hope for a perfect universal language, but in reality certain languages are better fitted for certain niches. For example, C and C++ remain popular for building operating systems and various other embedded systems because of their small runtime footprint and in spite of their lack of programming safety. This lack of safety can lead to programs crashing unpredictably and exposing security holes for viruses and the like; indeed, type-safe languages such as Java and C# have supplanted C and C++ in various applications when the additional runtime footprint is acceptable.

Prior occupancy of a niche tends to discourage competitors. Changing to a new language and tool chain is often too painful for just a single feature, but newcomers will eventually displace existing languages, unless they evolve fast enough to keep up. (Older readers are often able to quote a range of such languages in which they've previously coded but whose popularity has since waned—Ada, Algol, COBOL, Pascal, Delphi, and SNOBOL, to name but a few.)

You're a Java programmer, and Java has been successful at colonizing (and displacing competitor languages in) a large ecosystem niche of programming tasks for nearly 20 years. Let's examine some reasons for that.

### 1.2.1 Java's place in the programming language ecosystem

Java started well. Right from the start, it was a well-designed object-oriented language with many useful libraries. It also supported small-scale concurrency from day one with its integrated support for threads and locks (and with its early prescient acknowledgment, in the form of a hardware-neutral memory model, that concurrent threads on multicore processors can have unexpected behaviors in addition to those that happen on single-core processors). Also, the decision to compile Java to JVM bytecode (a virtual machine code that soon every browser supported) meant that it became the language of choice for internet applet programs (do you remember applets?). Indeed, there's a danger that the Java Virtual Machine (JVM) and its bytecode will be seen as more important than the Java language itself and that, for certain applications, Java might be replaced by one of its competing languages such as Scala, Groovy, or Kotlin, which also run on the JVM. Various recent updates to the JVM (for example, the new `invokedynamic` bytecode in JDK7) aim to help such competitor languages

---

<sup>2</sup> P. J. Landin, "The Next 700 Programming Languages," CACM 9(3):157–65, March 1966.



run smoothly on the JVM—and to interoperate with Java. Java has also been successful at colonizing various aspects of embedded computing (everything from smart cards, toasters, and set-top boxes to car-braking systems).

### How did Java get into a general programming niche?

Object orientation became fashionable in the 1990s for two reasons: its encapsulation discipline resulted in fewer software engineering issues than those of C; and as a mental model it easily captured the WIMP programming model of Windows 95 and up. This can be summarized as follows: everything is an object; and a mouse click sends an event message to a handler (invokes the `clicked` method in a `Mouse` object). The write-once, run-anywhere model of Java and the ability of early browsers to (safely) execute Java code applets gave it a niche in universities, whose graduates then populated industry. There was initial resistance to the additional run cost of Java over C/C++, but machines got faster, and programmer time became more and more important. Microsoft's C# further validated the Java-style object-oriented model.

But the climate is changing for the programming language ecosystem; programmers are increasingly dealing with so-called *big data* (data sets of terabytes and up) and wishing to exploit multicore computers or computing clusters effectively to process it. And this means using parallel processing—something Java wasn't previously friendly to. You may have come across ideas from other programming niches (for example, Google's map-reduce or the relative ease of data manipulation using database query languages such as SQL) that help you work with large volumes of data and multicore CPUs. Figure 1.1 summarizes the language ecosystem pictorially: think of the landscape as the space of programming problems and the dominant vegetation for a particular bit of ground as the favorite language for that program. Climate change is the idea that new hardware or new programming influences (for example, "Why can't I program in an SQL-like style?") mean that different languages become the language

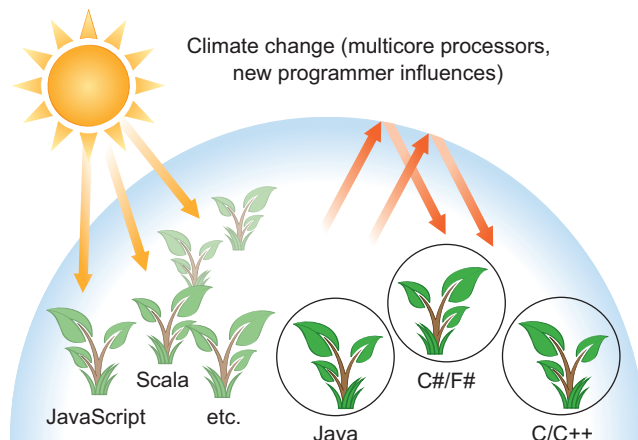


Figure 1.1 Programming-language ecosystem and climate change

of choice for new projects, just like increasing regional temperatures mean grapes now thrive in higher latitudes. But there's hysteresis—many an old farmer will keep raising traditional crops. In summary, new languages are appearing and becoming increasingly popular because they've adapted quickly to the climate change.

The main benefit of the Java 8 additions for a programmer is that they provide more programming tools and concepts to solve new or existing programming problems more quickly or, more importantly, in a more concise, more easily maintainable way. Although the concepts are new to Java, they've proved powerful in niche research-like languages. In the following sections, we'll highlight and develop the ideas behind three such programming concepts that have driven the development of the Java 8 features to exploit parallelism and write more concise code in general. We'll introduce them in a slightly different order from the rest of the book to enable a Unix-based analogy and to expose the “need *this* because of *that*” dependencies in Java 8's new parallelism for multicore.

### **Another climate-change factor for Java**

One climate-change factor involves how large systems are designed. Nowadays, it's common for a large system to incorporate large component subsystems from elsewhere, and perhaps these are built on top of other components from other vendors. Worse still, these components and their interfaces also tend to evolve. Java 8 and Java 9 have addressed these aspects by providing default methods and modules to facilitate this design style.

The next three sections examine the three programming concepts that drove the design of Java 8.

## **1.2.2 Stream processing**

The first programming concept is *stream processing*. For introductory purposes, a *stream* is a sequence of data items that are conceptually produced one at a time. A program might read items from an input stream one by one and similarly write items to an output stream. The output stream of one program could well be the input stream of another.

One practical example is in Unix or Linux, where many programs operate by reading data from standard input (*stdin* in Unix and C, *System.in* in Java), operating on it, and then writing their results to standard output (*stdout* in Unix and C, *System.out* in Java). First, a little background: Unix *cat* creates a stream by concatenating two files, *tr* translates the characters in a stream, *sort* sorts lines in a stream, and *tail -3* gives the last three lines in a stream. The Unix command line allows such programs to be linked together with pipes (*|*), giving examples such as

```
cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3
```

which (supposing `file1` and `file2` contain a single word per line) prints the three words from the files that appear latest in dictionary order, after first translating them to lowercase. We say that `sort` takes a *stream* of lines<sup>3</sup> as input and produces another stream of lines as output (the latter being sorted), as illustrated in figure 1.2. Note that in Unix these commands (`cat`, `tr`, `sort`, and `tail`) are executed concurrently, so that `sort` can be processing the first few lines before `cat` or `tr` has finished. A more mechanical analogy is a car-manufacturing assembly line where a stream of cars is queued between processing stations that each take a car, modify it, and pass it on to the next station for further processing; processing at separate stations is typically concurrent even though the assembly line is physically a sequence.

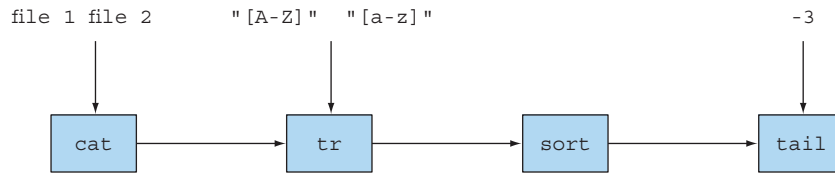


Figure 1.2 Unix commands operating on streams

Java 8 adds a Streams API (note the uppercase *S*) in `java.util.stream` based on this idea; `Stream<T>` is a sequence of items of type `T`. You can think of it as a fancy iterator for now. The Streams API has many methods that can be chained to form a complex pipeline just like Unix commands were chained in the previous example.

The key motivation for this is that you can now program in Java 8 at a higher level of abstraction, structuring your thoughts of turning a stream of this into a stream of that (similar to how you think when writing database queries) rather than one item at a time. Another advantage is that Java 8 can transparently run your pipeline of `Stream` operations on several CPU cores on disjoint parts of the input—this is parallelism *almost for free* instead of hard work using `Threads`. We cover the Java 8 Streams API in detail in chapters 4–7.

### 1.2.3 Passing code to methods with behavior parameterization

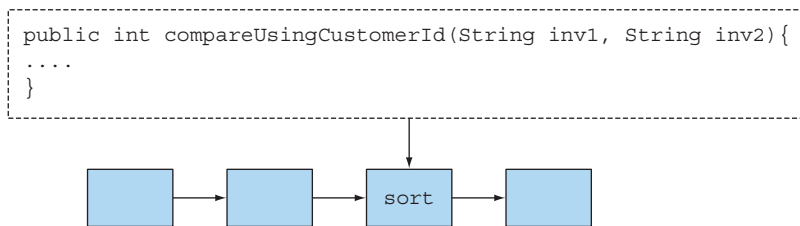
The second programming concept added to Java 8 is the ability to pass a piece of code to an API. This sounds awfully abstract. In the Unix example, you might want to tell the `sort` command to use a custom ordering. Although the `sort` command supports command-line parameters to perform various predefined kinds of sorting such as reverse order, these are limited.

For example, let's say you have a collection of invoice IDs with a format similar to 2013UK0001, 2014US0002, and so on. The first four digits represent the year, the next two letters a country code, and last four digits the ID of a client. You may want to sort

<sup>3</sup> Purists will say a “stream of characters,” but it's conceptually simpler to think that `sort` reorders *lines*.

these invoice IDs by year or perhaps using the customer ID or even the country code. What you want is the ability to tell the `sort` command to take as an argument an ordering defined by the user: a separate piece of code passed to the `sort` command.

Now, as a direct parallel in Java, you want to tell a `sort` method to compare using a customized order. You could write a method `compareUsingCustomerId` to compare two invoice IDs, but, prior to Java 8, you couldn't pass this method to another method! You could create a `Comparator` object to pass to the `sort` method as we showed at the start of this chapter, but this is verbose and obfuscates the idea of simply reusing an existing piece of behavior. Java 8 adds the ability to pass methods (your code) as arguments to other methods. Figure 1.3, based on figure 1.2, illustrates this idea. We also refer to this conceptually as *behavior parameterization*. Why is this important? The Streams API is built on the idea of passing code to parameterize the behavior of its operations, just as you passed `compareUsingCustomerId` to parameterize the behavior of `sort`.



**Figure 1.3** Passing method `compareUsingCustomerId` as an argument to `sort`

We summarize how this works in section 1.3 of this chapter, but leave full details to chapters 2 and 3. Chapters 18 and 19 look at more advanced things you can do using this feature, with techniques from the *functional programming* community.

### 1.2.4 *Parallelism and shared mutable data*

The third programming concept is rather more implicit and arises from the phrase “parallelism almost for free” in our previous discussion on stream processing. What do you have to give up? You may have to make some small changes in the way you code the behavior passed to stream methods. At first, these changes might feel a little uncomfortable, but once you get used to them, you’ll love them. You must provide behavior that *is safe to execute* concurrently on different pieces of the input. Typically this means writing code that doesn’t access shared mutable data to do its job. Sometimes these are referred to as pure functions or side-effect-free functions or stateless functions, and we’ll discuss these in detail in chapters 18 and 19. The previous parallelism arises only by assuming that multiple copies of your piece of code can work independently. If there’s a shared variable or object, which is written to, then things no longer work. What if two processes want to modify the shared variable at the same

time? (Section 1.4 gives a more detailed explanation with a diagram.) You'll find more about this style throughout the book.

Java 8 streams exploit parallelism more easily than Java's existing Threads API, so although it's *possible* to use `synchronized` to break the no-shared-mutable-data rule, it's fighting the system in that it's abusing an abstraction optimized around that rule. Using `synchronized` across multiple processing cores is often far more expensive than you expect, because synchronization forces code to execute sequentially, which works against the goal of parallelism.

Two of these points (no shared mutable data and the ability to pass methods and functions—code—to other methods) are the cornerstones of what's generally described as the paradigm of *functional programming*, which you'll see in detail in chapters 18 and 19. In contrast, in the *imperative programming* paradigm you typically describe a program in terms of a sequence of statements that mutate state. The no-shared-mutable-data requirement means that a method is perfectly described solely by the way it transforms arguments to results; in other words, it behaves as a mathematical function and has no (visible) side effects.

### 1.2.5 Java needs to evolve

You've seen evolution in Java before. For example, the introduction of generics and using `List<String>` instead of just `List` may initially have been irritating. But you're now familiar with this style and the benefits it brings (catching more errors at compile time and making code easier to read, because you now know what something is a list of).

Other changes have made common things easier to express (for example, using a for-each loop instead of exposing the boilerplate use of an `Iterator`). The main changes in Java 8 reflect a move away from classical object orientation, which often focuses on mutating existing values, and toward the functional-style programming spectrum in which *what* you want to do in broad-brush terms (for example, *create a value* representing all transport routes from A to B for less than a given price) is considered prime and separated from *how* you can achieve this (for example, *scan* a data structure *modifying* certain components). Note that classical object-oriented programming and functional programming, as extremes, might appear to be in conflict. But the idea is to get the best from both programming paradigms, so you have a better chance of having the right tool for the job. We discuss this in detail in sections 1.3 and 1.4.

A takeaway line might be this: languages need to evolve to track changing hardware or programmer expectations (if you need convincing, consider that COBOL was once one of the most important languages commercially). To endure, Java has to evolve by adding new features. This evolution will be pointless unless the new features are used, so in using Java 8 you're protecting your way of life as a Java programmer. On top of that, we have a feeling you'll love using Java 8's new features. Ask anyone who's used Java 8 whether they're willing to go back! Additionally, the new Java 8

features might, in the ecosystem analogy, enable Java to conquer programming-task territory currently occupied by other languages, so Java 8 programmers will be even more in demand.

We now introduce the new concepts in Java 8, one by one, pointing out the chapters that cover these concepts in more detail.

## 1.3 *Functions in Java*

The word *function* in programming languages is commonly used as a synonym for *method*, particularly a static method; this is in addition to it being used for *mathematical function*, one without side effects. Fortunately, as you'll see, when Java 8 refers to functions these usages nearly coincide.

Java 8 adds functions as new forms of value. These facilitate the use of streams, covered in section 1.4, which Java 8 provides to exploit parallel programming on multi-core processors. We start by showing that functions as values are useful in themselves.

Think about the possible values manipulated by Java programs. First, there are primitive values such as 42 (of type `int`) and 3.14 (of type `double`). Second, values can be objects (more strictly, references to objects). The only way to get one of these is by using `new`, perhaps via a factory method or a library function; object references point to *instances* of a class. Examples include `"abc"` (of type `String`), `new Integer(1111)` (of type `Integer`), and the result `new HashMap<Integer, String>(100)` of explicitly calling a constructor for `HashMap`. Even arrays are objects. What's the problem?

To help answer this, we'll note that the whole point of a programming language is to manipulate values, which, following historical programming-language tradition, are therefore called first-class values (or citizens, in the terminology borrowed from the 1960s civil rights movement in the United States). Other structures in our programming languages, which perhaps help us express the structure of values but which can't be passed around during program execution, are second-class citizens. Values as listed previously are first-class Java citizens, but various other Java concepts, such as methods and classes, exemplify second-class citizens. Methods are fine when used to define classes, which in turn may be instantiated to produce values, but neither are values themselves. Does this matter? Yes, it turns out that being able to pass methods around at runtime, and hence making them first-class citizens, is useful in programming, so the Java 8 designers added the ability to express this directly in Java. Incidentally, you might wonder whether making other second-class citizens such as classes into first-class-citizen values might also be a good idea. Various languages such as Smalltalk and JavaScript have explored this route.

### 1.3.1 *Methods and lambdas as first-class citizens*

Experiments in other languages, such as Scala and Groovy, have determined that allowing concepts like methods to be used as first-class values made programming easier by adding to the toolset available to programmers. And once programmers become familiar with a powerful feature, they become reluctant to use languages

without it! The designers of Java 8 decided to allow methods to be values—to make it easier for you to program. Moreover, the Java 8 feature of methods as values forms the basis of various other Java 8 features (such as Streams).

The first new Java 8 feature we introduce is that of *method references*. Suppose you want to filter all the hidden files in a directory. You need to start writing a method that, given a `File`, will tell you whether it’s hidden. Fortunately, there’s such a method in the `File` class called `isHidden`. It can be viewed as a function that takes a `File` and returns a boolean. But to use it for filtering, you need to wrap it into a `FileFilter` object that you then pass to the `File.listFiles` method, as follows:

```
File[] hiddenFiles = new File(".").listFiles(new FileFilter() {
    public boolean accept(File file) {
        return file.isHidden();
    }
});
```

← Filtering hidden files!

Yuck! That’s horrible. Although it’s only three significant lines, it’s three opaque lines—we all remember saying “Do I really have to do it this way?” on first encounter. You already have the method `isHidden` that you could use. Why do you have to wrap it up in a verbose `FileFilter` class and then instantiate it? Because that’s what you had to do prior to Java 8.

Now, you can rewrite that code as follows:

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

Wow! Isn’t that cool? You already have the function `isHidden` available, so you pass it to the `listFiles` method using the Java 8 *method reference* `::` syntax (meaning “use this method as a value”); note that we’ve also slipped into using the word *function* for methods. We’ll explain later how the mechanics work. One advantage is that your code now reads closer to the problem statement.

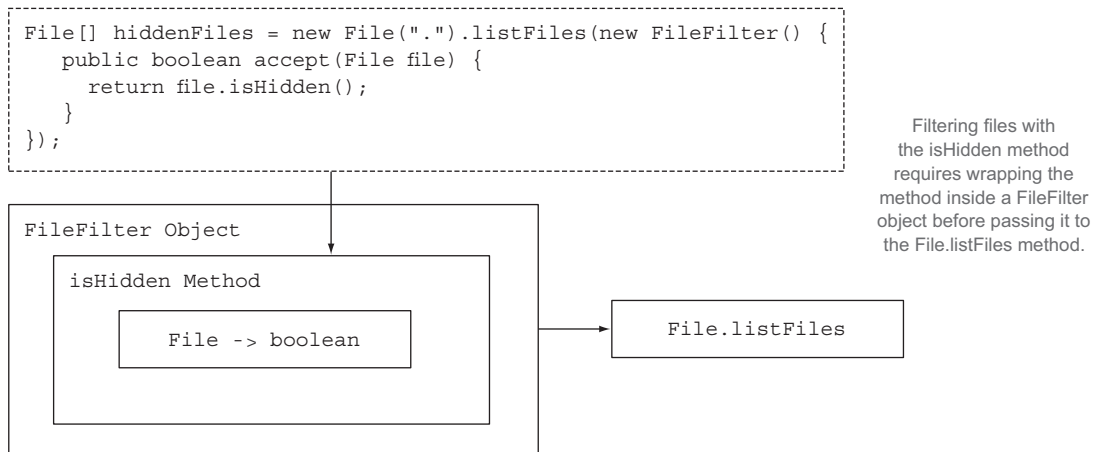
Here’s a taste of what’s coming: methods are no longer second-class values. Analogous to using an *object reference* when you pass an object around (and object references are created by `new`), in Java 8 when you write `File::isHidden`, you create a *method reference*, which can similarly be passed around. This concept is discussed in detail in chapter 3. Given that methods contain code (the executable body of a method), using method references enables passing code around as in figure 1.3. Figure 1.4 illustrates the concept. You’ll also see a concrete example (selecting apples from an inventory) in the next section.

### LAMBDA: ANONYMOUS FUNCTIONS

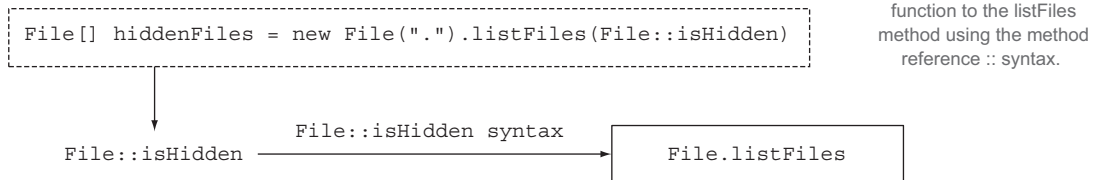
As well as allowing (named) methods to be first-class values, Java 8 allows a richer idea of *functions as values*, including *lambdas*<sup>4</sup> (or anonymous functions). For example, you can now write `(int x) -> x + 1` to mean “the function that, when called with argument

<sup>4</sup> Originally named after the Greek letter  $\lambda$  (lambda). Although the symbol isn’t used in Java, its name lives on.

## Old way of filtering hidden files



## Java 8 style



**Figure 1.4** Passing the method reference `File::isHidden` to the method `listFiles`

`x`, returns the value `x + 1`.” You might wonder why this is necessary, because you could define a method `add1` inside a class `MyMathsUtils` and then write `MyMathsUtils::add1`! Yes, you could, but the new lambda syntax is more concise for cases where you don’t have a convenient method and class available. Chapter 3 explores lambdas in detail. Programs using these concepts are said to be written in functional-programming style; this phrase means “writing programs that pass functions around as first-class values.”

### 1.3.2 Passing code: an example

Let’s look at an example of how this helps you write programs (discussed in more detail in chapter 2). All the code for the examples is available on a GitHub repository and as a download via the book’s website. Both links may be found at [www.manning.com/books/modern-java-in-action](http://www.manning.com/books/modern-java-in-action). Suppose you have a class `Apple` with a method `getColor` and a variable `inventory` holding a list of `Apples`; then you might wish to select all the green apples (here using a `Color` enum type that includes values `GREEN`



and RED) and return them in a list. The word *filter* is commonly used to express this concept. Before Java 8, you might write a method `filterGreenApples`:

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if (GREEN.equals(apple.getColor())) {
            result.add(apple);
        }
    }
    return result;
}
```

The highlighted text selects only green apples.

The result list accumulates the result; it starts as empty, and then green apples are added one by one.

But next, somebody would like the list of heavy apples (say over 150 g), and so, with a heavy heart, you'd write the following method to achieve this (perhaps even using copy and paste):

```
public static List<Apple> filterHeavyApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if (apple.getWeight() > 150) {
            result.add(apple);
        }
    }
    return result;
}
```

Here the highlighted text selects only heavy apples.

We all know the dangers of copy and paste for software engineering (updates and bug fixes to one variant but not the other), and hey, these two methods vary only in one line: the highlighted condition inside the `if` construct. If the difference between the two method calls in the highlighted code had been what weight range was acceptable, then you could have passed lower and upper acceptable weights as arguments to `filter`—perhaps (150, 1000) to select heavy apples (over 150 g) or (0, 80) to select light apples (under 80 g).

But as we mentioned previously, Java 8 makes it possible to pass the code of the condition as an argument, avoiding code duplication of the `filter` method. You can now write this:

```
public static boolean isGreenApple(Apple apple) {
    return GREEN.equals(apple.getColor());
}
public static boolean isHeavyApple(Apple apple) {
    return apple.getWeight() > 150;
}
public interface Predicate<T>{
    boolean test(T t);
}
static List<Apple> filterApples(List<Apple> inventory,
                               Predicate<Apple> p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
```

Included for clarity (normally imported from `java.util.function`)

A method is passed as a Predicate parameter named `p` (see the sidebar “What’s a Predicate?”).

```

        if (p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}

```

← Does the apple match the condition represented by p?

And to use this, you call either

```
filterApples(inventory, Apple::isGreenApple);
```

or

```
filterApples(inventory, Apple::isHeavyApple);
```

We explain how this works in detail in the next two chapters. The key idea to take away for now is that you can pass around a method in Java 8.

### What's a Predicate?

The previous code passed a method `Apple::isGreenApple` (which takes an `Apple` for argument and returns a `boolean`) to `filterApples`, which expected a `Predicate<Apple>` parameter. The word *predicate* is often used in mathematics to mean something function-like that takes a value for an argument and returns `true` or `false`. As you'll see later, Java 8 would also allow you to write `Function<Apple, Boolean>`—more familiar to readers who learned about functions but not predicates at school—but using `Predicate<Apple>` is more standard (and slightly more efficient because it avoids boxing a `boolean` into a `Boolean`).

### 1.3.3 From passing methods to lambdas

Passing methods as values is clearly useful, but it's annoying having to write a definition for short methods such as `isHeavyApple` and `isGreenApple` when they're used perhaps only once or twice. But Java 8 has solved this, too. It introduces a new notation (anonymous functions, or *lambdas*) that enables you to write just

```
filterApples(inventory, (Apple a) -> GREEN.equals(a.getColor()));
```

or

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150);
```

or even

```
filterApples(inventory, (Apple a) -> a.getWeight() < 80 ||
    RED.equals(a.getColor()));
```

You don't even need to write a method definition that's used only once; the code is crisper and clearer because you don't need to search to find the code you're passing.

But if such a lambda exceeds a few lines in length (so that its behavior isn't instantly clear), you should instead use a method reference to a method with a descriptive name instead of using an anonymous lambda. Code clarity should be your guide.

The Java 8 designers could almost have stopped here, and perhaps they would have done so before multicore CPUs. Functional-style programming as presented so far turns out to be powerful, as you'll see. Java might then have been rounded off by adding `filter` and a few friends as generic library methods, such as

```
static <T> Collection<T> filter(Collection<T> c, Predicate<T> p);
```

You wouldn't even have to write methods like `filterApples` because, for example, the previous call

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

could be written as a call to the library method `filter`:

```
filter(inventory, (Apple a) -> a.getWeight() > 150 );
```

But, for reasons centered on better exploiting parallelism, the designers didn't do this. Java 8 instead contains a new Collection-like API called `Stream`, containing a comprehensive set of operations similar to the `filter` operation that functional programmers may be familiar with (for example, `map` and `reduce`), along with methods to convert between `Collections` and `Streams`, which we now investigate.

## 1.4 Streams


Nearly every Java application *makes* and *processes* collections. But working with collections isn't always ideal. For example, let's say you need to filter expensive transactions from a list and then group them by currency. You'd need to write a lot of boilerplate code to implement this data-processing query, as shown here:



In addition, it's difficult to understand at a glance what the code does because of the multiple nested control-flow statements.

Using the Streams API, you can solve this problem as follows:

```
import static java.util.stream.Collectors.groupingBy;
Map<Currency, List<Transaction>> transactionsByCurrencies =
    transactions.stream()
        .filter((Transaction t) -> t.getPrice() > 1000)
        .collect(groupingBy(Transaction::getCurrency));
```



Don't worry about this code for now because it may look like a bit of magic. Chapters 4–7 are dedicated to explaining how to make sense of the Streams API. For now, it's worth noticing that the Streams API provides a different way to process data in comparison to the Collections API. Using a collection, you're managing the iteration process yourself. You need to iterate through the elements one by one using a `for-each` loop processing them in turn. We call this way of iterating over data *external iteration*. In contrast, using the Streams API, you don't need to think in terms of loops. The data processing happens internally inside the library. We call this idea *internal iteration*. We come back to these ideas in chapter 4.

As a second pain point of working with collections, think for a second about how you would process the list of transactions if you had a vast number of them; how can you process this huge list? A single CPU wouldn't be able to process this large amount of data, but you probably have a multicore computer on your desk. Ideally, you'd like to share the work among the different CPU cores available on your machine to reduce the processing time. In theory, if you have eight cores, they should be able to process your data eight times as fast as using one core, because they work in parallel.<sup>5</sup>

### Multicore computers

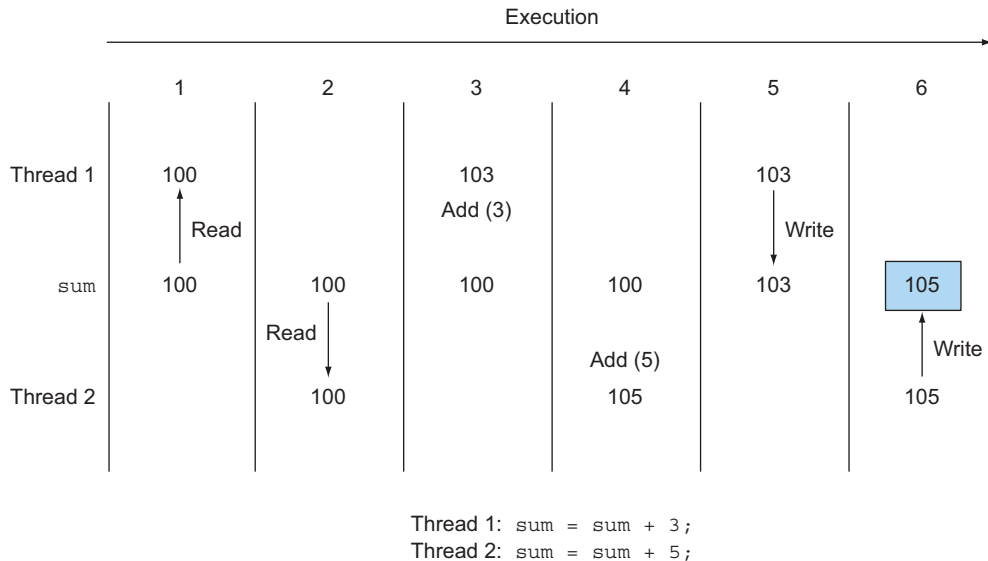
All new desktop and laptop computers are multicore computers. Instead of a single CPU, they have four or eight or more CPUs (usually called Cores5). The problem is that a classic Java program uses just a single one of these cores, and the power of the others is wasted. Similarly, many companies use *computing clusters* (computers connected together with fast networks) to be able to process vast amounts of data efficiently. Java 8 facilitates new programming styles to better exploit such computers.

Google's search engine is an example of a piece of code that's too big to run on a single computer. It reads every page on the internet and creates an index, mapping every word appearing on any internet page back to every URL containing that word. Then, when you do a Google search involving several words, software can quickly use this index to give you a set of web pages containing those words. Try to imagine how you might code this algorithm in Java (even for a smaller index than Google's, you'd need to exploit all the cores in your computer).

<sup>5</sup> This naming is unfortunate in some ways. Each of the cores in a multicore chip is a full-fledged CPU. But the phrase multicore CPU has become common, so core is used to refer to the individual CPUs.

### 1.4.1 Multithreading is difficult

The problem is that exploiting parallelism by writing *multithreaded* code (using the Threads API from previous versions of Java) is difficult. You have to think differently: threads can access and update shared variables at the same time. As a result, data could change unexpectedly if not coordinated<sup>6</sup> properly. This model is harder to think about<sup>7</sup> than a step-by-step sequential model. For example, figure 1.5 shows a possible problem with two threads trying to add a number to a shared variable `sum` if they're not synchronized properly.



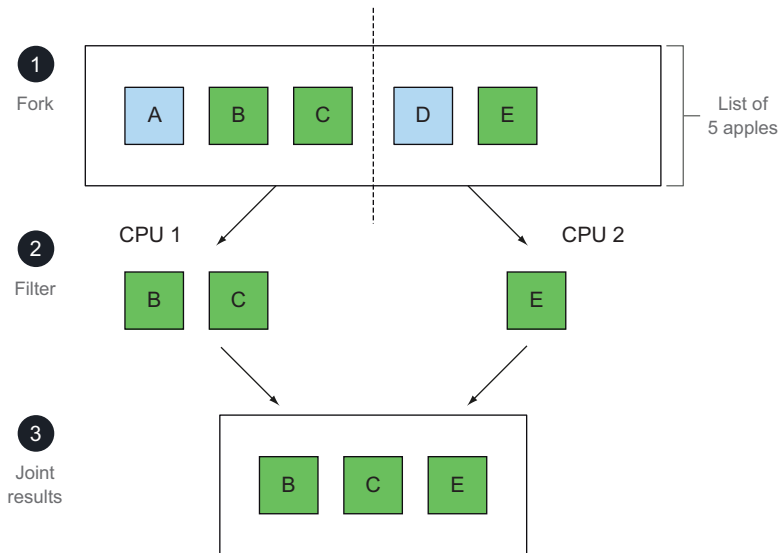
**Figure 1.5** A possible problem with two threads trying to add to a shared `sum` variable. The result is 105 instead of an expected result of 108.

Java 8 also addresses both problems (boilerplate and obscurity involving processing collections and difficulty exploiting multicore) with the Streams API (`java.util.stream`). The first design motivator is that there are many data-processing patterns (similar to `filterApples` in the previous section or operations familiar from database query languages such as SQL) that occur over and over again and that would benefit from forming part of a library: *filtering* data based on a criterion (for example, heavy apples), *extracting* data (for example, extracting the weight field from each apple in a list), or *grouping* data (for example, grouping a list of numbers into separate lists of even and odd numbers), and so on. The second motivator is that such operations can

<sup>6</sup> Traditionally via the keyword `synchronized`, but many subtle bugs arise from its misplacement. Java 8's Stream-based parallelism encourages a functional programming style where `synchronized` is rarely used; it focuses on partitioning the data rather than coordinating access to it.

<sup>7</sup> Aha—a source of pressure for the language to evolve!

often be parallelized. For instance, as illustrated in figure 1.6, filtering a list on two CPUs could be done by asking one CPU to process the first half of a list and the second CPU to process the other half of the list. This is called the *forking step* (1). The CPUs then filter their respective half-lists (2). Finally (3), one CPU would join the two results. (This is closely related to how Google searches work so quickly, using many more than two processors.)



**Figure 1.6** Forking filter onto two CPUs and joining the result

For now, we'll just say that the new Streams API behaves similarly to Java's existing Collections API: both provide access to sequences of data items. But it's useful for now to keep in mind that Collections is mostly about storing and accessing data, whereas Streams is mostly about describing computations on data. The key point here is that the Streams API allows and encourages the elements within a stream to be processed in parallel. Although it may seem odd at first, often the fastest way to filter a collection (for example, to use `filterApples` in the previous section on a list) is to convert it to a stream, process it in parallel, and then convert it back to a list. Again, we'll just say "parallelism almost for free" and provide a taste of how you can filter heavy apples from a list sequentially or in parallel using streams and a lambda expression.

Here's an example of sequential processing:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                .collect(toList());
```

And here it is using parallel processing:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                .collect(toList());
```

### Parallelism in Java and no shared mutable state

People have always said parallelism in Java is difficult, and all this stuff about synchronized is error-prone. Where's the magic bullet in Java 8?

There are two magic bullets. First, the library handles partitioning—breaking down a big stream into several smaller streams to be processed in parallel for you. Second, this parallelism almost for free from streams, works only if the methods passed to library methods like `filter` don't interact (for example, by having mutable shared objects). But it turns out that this restriction feels natural to a coder (see, by way of example, our `Apple::isGreenApple` example). Although the primary meaning of *functional* in *functional programming* means “using functions as first-class values,” it often has a secondary nuance of “no interaction during execution between components.”

Chapter 7 explores parallel data processing in Java 8 and its performance in more detail. One of the practical issues the Java 8 developers found in evolving Java with all these new goodies was that of evolving existing interfaces. For example, the method `Collections.sort` belongs to the `List` interface but was never included. Ideally, you'd like to do `list.sort(comparator)` instead of `Collections.sort(list, comparator)`. This may seem trivial but, prior to Java 8 you can update an interface only if you update all the classes that implement it—a logistical nightmare! This issue is resolved in Java 8 by *default methods*.

## 1.5 Default methods and Java modules

As we mentioned earlier, modern systems tend to be built from components—perhaps bought-in from elsewhere. Historically, Java had little support for this, apart from a JAR file containing a set of Java packages with no particular structure. Moreover, evolving interfaces to such packages was hard—changing a Java interface meant changing every class that implements it. Java 8 and 9 have started to address this.

First, Java 9 provides a module system that provide you with syntax to define *modules* containing collections of packages—and keep much better control over visibility and namespaces. Modules enrich a simple JAR-like component with structure, both as user documentation and for machine checking; we explain them in detail in chapter 14. Second, Java 8 added default methods to support *evolvable* interfaces. We cover these in detail in chapter 13. They're important because you'll increasingly encounter them in interfaces, but because relatively few programmers will need to write default methods themselves and because they facilitate program evolution

rather than helping write any particular program, we keep the explanation here short and example-based.

In section 1.4, we gave the following example Java 8 code:

```
List<Apple> heavyApples1 =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                .collect(toList());
List<Apple> heavyApples2 =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                .collect(toList());
```

But there's a problem here: a `List<T>` prior to Java 8 doesn't have `stream` or `parallelStream` methods—and neither does the `Collection<T>` interface that it implements—because these methods hadn't been conceived of. And without these methods, this code won't compile. The simplest solution, which you might employ for your own interfaces, would have been for the Java 8 designers to add the `stream` method to the `Collection` interface and add the implementation in the `ArrayList` class.

But doing this would have been a nightmare for users. Many alternative collection frameworks implement interfaces from the Collections API. Adding a new method to an interface means all concrete classes must provide an implementation for it. Language designers have no control over existing implementations of `Collection`, so you have a dilemma: How can you evolve published interfaces without disrupting existing implementations?

The Java 8 solution is to break the last link: an interface can now contain method signatures for which an implementing class doesn't provide an implementation. Then who implements them? The missing method bodies are given as part of the interface (hence default implementations) rather than in the implementing class.

This provides a way for an interface designer to enlarge an interface beyond those methods that were originally planned—without breaking existing code. Java 8 allows the existing `default` keyword to be used in interface specifications to achieve this.

For example, in Java 8, you can call the `sort` method directly on a list. This is made possible with the following default method in the Java 8 `List` interface, which calls the static method `Collections.sort`:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

This means any concrete classes of `List` don't have to explicitly implement `sort`, whereas in previous Java versions such concrete classes would fail to recompile unless they provided an implementation for `sort`.

But wait a second. A single class can implement multiple interfaces, right? If you have multiple default implementations in several interfaces, does that mean you have a form of multiple inheritance in Java? Yes, to some extent. We show in chapter 13 that there are some rules that prevent issues such as the infamous *diamond inheritance problem* in C++.



## 1.6 Other good ideas from functional programming

The previous sections introduced two core ideas from functional programming that are now part of Java: using methods and lambdas as first-class values, and the idea that calls to functions or methods can be efficiently and safely executed in parallel in the absence of mutable shared state. Both of these ideas are exploited by the new Streams API we described earlier.

Common functional languages (SML, OCaml, Haskell) also provide further constructs to help programmers. One of these is avoiding null by explicit use of more descriptive data types. Tony Hoare, one of the giants of computer science, said this in a presentation at QCon London 2009:

*I call it my billion-dollar mistake. It was the invention of the null reference in 1965... I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.*

Java 8 introduced the `Optional<T>` class that, if used consistently, can help you avoid null-pointer exceptions. It's a container object that may or may not contain a value. `Optional<T>` includes methods to explicitly deal with the case where a value is absent, and as a result you can avoid null-pointer exceptions. It uses the type system to allow you to indicate when a variable is anticipated to potentially have a missing value. We discuss `Optional<T>` in detail in chapter 11.

A second idea is that of (*structural*) *pattern matching*.<sup>8</sup> This is used in mathematics. For example:

$$\begin{aligned} f(0) &= 1 \\ f(n) &= n * f(n-1) \text{ otherwise} \end{aligned}$$

In Java, you would write an `if-then-else` or a `switch` statement. Other languages have shown that, for more complex data types, pattern matching can express programming ideas more concisely compared to using `if-then-else`. For such data types, you might also use polymorphism and method overriding as an alternative to `if-then-else`, but there's ongoing language-design discussion as to which is more appropriate.<sup>9</sup> We'd say that both are useful tools and that you should have both in your armory. Unfortunately, Java 8 doesn't have full support for pattern matching, although we show how it can be expressed in chapter 19. A Java Enhancement Proposal is also being discussed to support pattern matching in a future version of Java (see <http://openjdk.java.net/jeps/305>). In the meantime, let's illustrate with an example expressed in the Scala programming language (another Java-like language using the JVM that has inspired some aspects of Java evolution; see chapter 20). Suppose you

<sup>8</sup> This phrase has two uses. Here we mean the one familiar from mathematics and functional programming whereby a function is defined by cases, rather than using `if-then-else`. The other meaning concerns phrases like "find all files of the form 'IMG\*.JPG' in a given directory" associated with so-called regular expressions.

<sup>9</sup> The Wikipedia article on the "expression problem" (a term coined by Phil Wadler) provides an entry to the discussion.

want to write a program that does basic simplifications on a tree representing an arithmetic expression. Given a data type `Expr` representing such expressions, in Scala you can write the following code to decompose an `Expr` into its parts and then return another `Expr`:

```
def simplifyExpression(expr: Expr): Expr = expr match {
  case BinOp("+", e, Number(0)) => e
  case BinOp("-", e, Number(0)) => e
  case BinOp("*", e, Number(1)) => e
  case BinOp("/", e, Number(1)) => e
  case _ => expr
}
```

Here Scala's syntax `expr match` corresponds to Java's `switch (expr)`. Don't worry about this code for now—you'll read more on pattern matching in chapter 19. For now, you can think of pattern matching as an extended form of `switch` that can decompose a data type into its components at the same time.

Why should the `switch` statement in Java be limited to primitive values and strings? Functional languages tend to allow `switch` to be used on many more data types, including allowing pattern matching (in the Scala code, this is achieved using a `match` operation). In object-oriented design, the visitor pattern is a common pattern used to walk through a family of classes (such as the different components of a car: wheel, engine, chassis, and so on) and apply an operation to each object visited. One advantage of pattern matching is that a compiler can report common errors such as, "Class Brakes is part of the family of classes used to represent components of class Car. You forgot to explicitly deal with it."

Chapters 18 and 19 give a full tutorial introduction to functional programming and how to write functional-style programs in Java 8—including the toolkit of functions provided in its library. Chapter 20 follows by discussing how Java 8 features compare to those in Scala—a language that, like Java, is implemented on top of the JVM and that has evolved quickly to threaten some aspects of Java's niche in the programming language ecosystem. This material is positioned toward the end of the book to provide additional insight into why the new Java 8 and Java 9 features were added.

### Java 8, 9, 10, and 11 features: Where do you start?

Java 8 and Java 9 both provided significant updates to Java. But as a Java programmer, it's likely to be the Java 8 additions that affect you most on a daily small-scale-coding basis—the idea of passing a method or a lambda is rapidly becoming vital Java knowledge. In contrast, the Java 9 enhancements enrich our ability to define and use larger-scale components, be it structuring a system using modules or importing a reactive-programming toolkit. Finally, Java 10 is a much smaller increment compared to previous upgrades and consists of allowing type inference for local variables, which we discuss briefly in chapter 21, where we also mention the related richer syntax for arguments of lambda expressions due to be introduced in Java 11.

At the time of writing, Java 11 is scheduled to be released in September 2018. Java 11 also brings a new asynchronous HTTP client library (<http://openjdk.java.net/jeps/321>) that leverages the Java 8 and Java 9 developments (details in chapters 15, 16, and 17) of `CompletableFuture` and reactive programming.

## Summary

- Keep in mind the idea of the language ecosystem and the consequent evolve-or-wither pressure on languages. Although Java may be supremely healthy at the moment, we can recall other healthy languages such as COBOL that failed to evolve.
- The core additions to Java 8 provide exciting new concepts and functionality to ease the writing of programs that are both effective and concise.
- Multicore processors aren't fully served by pre-Java-8 programming practice.
- Functions are first-class values; remember how methods can be passed as functional values and how anonymous functions (lambdas) are written.
- The Java 8 concept of streams generalizes many aspects of collections, but the former often enables more readable code and allows elements of a stream to be processed in parallel.
- Large-scale component-based programming, and evolving a system's interfaces, weren't historically well served by Java. You can now specify modules to structure systems in Java 9 and use default methods to allow an interface to be enhanced without changing all the classes that implement it.
- Other interesting ideas from functional programming include dealing with `null` and using pattern matching.

# Modern Java IN ACTION

Urma • Fusco • Mycroft



**M**odern applications take advantage of innovative designs, including microservices, reactive architectures, and streaming data. Modern Java features like lambdas, streams, and the long-awaited Java Module System make implementing these designs significantly easier. It's time to upgrade your skills and meet these challenges head on!

**Modern Java in Action** connects new features of the Java language with their practical applications. Using crystal-clear examples and careful attention to detail, this book respects your time. It will help you expand your existing knowledge of core Java as you master modern additions like the Streams API and the Java Module System, explore new approaches to concurrency, and learn how functional concepts can help you write code that's easier to read and maintain.

## What's Inside

- Thoroughly revised edition of Manning's bestselling *Java 8 in Action*
- New features in Java 8, Java 9, and beyond
- Streaming data and reactive programming
- The Java Module System

Written for developers familiar with core Java features.

**Raoul-Gabriel Urma** is CEO of Cambridge Spark. **Mario Fusco** is a senior software engineer at Red Hat. **Alan Mycroft** is a University of Cambridge computer science professor; he cofounded the Raspberry Pi Foundation.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit [manning.com/books/modern-java-in-action](http://manning.com/books/modern-java-in-action)

“A comprehensive and practical introduction to the modern features of the latest Java releases with excellent examples!”

—Oleksandr Mandryk  
EPAM Systems

“Hands-on Java 8 and 9, simply and elegantly explained.”

—Deepak Bhaskaran, Salesforce

“A lot of great examples and use cases for streams, concurrency, and reactive programming.”

—Rob Pacheco, Synopsys

“My Java code improved significantly after reading this book. I was able to take the clear examples and immediately put them into practice.”

—Holly Cummins, IBM

ISBN-13: 978-1-61729-356-6  
ISBN-10: 1-61729-356-3



9 781617 293566



\$54.99 / Can \$72.99 [INCLUDING eBook]