Java 8 IN ACTION

Lambdas, Streams, and functional-style programming

Raoul-Gabriel Urma
Mario Fusco
Alan Mycroft

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Java 8 in Action: Lambdas, Streams and Functional-style Programming
Version 4.1

For more information on this and other Manning titles go to
www.manning.com

brief contents

1

Java 8: why should you care?

This chapter covers:

- Why are *they* changing Java again?
- Changing computing background: big data, multicore, cloud computing.
- Java pressured to evolve: new architectures favor functional-style over imperative
- Summary of new features in Java 8: lambda expressions, streams, default methods.

Since the release of JDK 1.0 (Java 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. And now Java 8 is to appear in 2014. So the question is, Why should you care about Java 8?

- Perhaps your boss has decreed that some new project will use Java 8 (and you wish people would just stop messing with languages and let you get on with your job as a Java programmer).
- Perhaps you've heard that Java 8 makes it easier to exploit more of the multiple CPUs on your desktop or laptop computer.
- Perhaps you've heard that Java 8 makes it easier to write in simple broad-brush strokes using higher-level ideas.

Even if you fall into the first category and just want to know what's new in Java 8, we beg your indulgence. We spend just two pages wondering why languages evolve and then turn to the tectonic forces causing the evolution to Java 8. These can be summarized as multicore CPUs, big data, and the fact that programmers have learned new programming techniques

using other languages and liked them, and want to use them in Java too (for example, why is so much easier to process data in databases using SQL than Java?).

We then introduce each new idea (passing code with behavior parameterization, lambdas, streams, and default methods) in a tutorial style—covering background so you can see why the developments happen, but also giving simple initial examples to be developed in more detail in successive chapters.

We argue that the changes to Java 8 are in many ways more profound than any other change to Java in its 18-year history.

The medium picture is that instead of writing code like this (to sort a list of apples based on their weight):

BEFORE

```
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:

AFTER

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

#A: first Java 8 code of the book!

Don't worry about this code for now. This book will explain you what it does and how you can write similar code!

However, the big picture is this: commodity CPUs have become multicore; the processor in your laptop or desktop machine probably has four or so CPU cores within it. But the vast majority of Java programs use only one of these cores and leave the other seven idle (or doing something unrelated such as running part of the operating system or as a virus checker).

The problem is that working with threads is hard. Java has followed an evolutionary path of trying to make concurrency easier and less error-prone. Java 1.0 had threads and locks, and even a memory model, but doing anything was still hard. Java 5 added industrial-strength building blocks like thread pools and concurrent collections. Java 7 added the fork-join framework, making parallelism practical, but still hard. Java 8 has a new, simpler way of thinking about parallelism (but you still have to follow some rules which you will learn in this book!). Java 8 provides a new API (called `Streams`) that supports many parallel operations to process data and resembles the way you might think in SQL. As result, it avoids the need for you to write code that uses `synchronized`, which is not only highly error prone but is also more expensive than programmers often realize on multicore CPUs. The addition of `Streams` in Java 8 can be seen as a direct cause of the two other additions to Java 8: *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 programmers a new way to express *behavior parameterization*. Suppose you want to write two methods that differ in only a few lines of code; you can now 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). This Java 8 feature also provides access to a whole range of further techniques, which are commonly referred to as *functional-style programming*. We start this chapter with a high-level discussion on why languages evolve, continue with sections on the core features of Java 8, and then introduce the ideas of functional-style programming, which the new features simplify using and which new computer architectures favor. In essence section 1.1 starts with the evolution process and the concepts, which Java was previously lacking, to exploit multicore parallelism in an easy way; section 1.2 explains why passing code to methods in Java 8 is such a powerful new programming idiom, and section 1.3 does the same for `Streams`—the new Java 8 way of representing sequenced data and processing it in parallel. Section 1.4 explains how the new Java 8 feature of default methods enables interfaces and their libraries to evolve with less fuss and less recompilation. Finally, section 1.5 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!

1.1 Why is Java still changing?

In the 1960s there was the quest for the perfect programming language. In a landmark article, Peter Landin, famous computer scientist of his day, noted1 in 1966 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 an ecosystem: new languages appear and old languages are supplanted unless they evolve. Programmers live in hope of 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, in spite of their lack of programming safety, leading to programs crashing unpredictably and opening security holes for viruses and the like.

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 will often be able to quote a range of such languages in which they may have previously coded but

1 P. J. Landin. The Next 700 Programming Languages. CACM 9(3):157–65, March 1966.

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 a large worldwide niche for the last 15 years. Let's examine some reasons for that.

1.1.1 Java's place in the 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 its formal cross-platform memory model. In addition, 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 and its bytecode will be seen as more important than the Java language itself, and that Java might be replaced by one of its competing languages such as Scala or Groovy, which also run on the Java virtual machine. In fact, many recent updates to the Java virtual machine (for example, the new *invokedynamic* bytecode in JDK7) aim to help such competitor languages run smoothly on the JVM.

Java has also been successful at colonizing various aspects of embedded computing (everything from smartcards, toasters, and set-top boxes to car braking systems and autopilots).

How did Java get into a general programming niche at all?

Object orientation had become fashionable in the 1990s for two reasons: (a) its encapsulation discipline resulted in fewer software engineering issues than those of C and (b) it matched the WIMP model of Windows 95 and up. Everything is an object; a mouse click sends an event message to a handler (invokes the `Clicked` method in a `Mouse` object). The write-once run-anywhere model and the ability of early browsers to execute (safe) 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 *big data* (terabytes and up) and wishing to exploit multicore computers or computing clusters effectively to process this. And this means using parallel processing—something Java was not previously good at.

You may have come across programming ideas from other programming niches (for example, Google's map-reduce or the relative ease of data manipulation in SQL) that help you work with big data and multicore CPUs. Figure 1.1 summarizes the language ecosystem

pictorially: new languages are appearing and becoming increasingly popular because they've adapted quickly to the climate change.

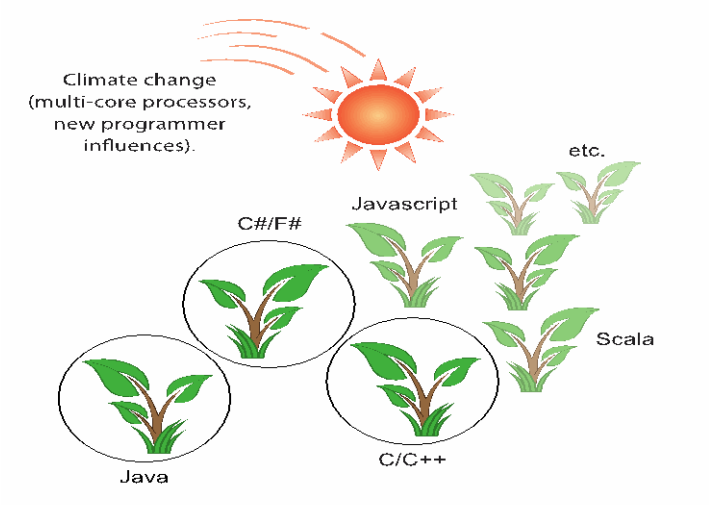


Figure 1.1 Programming languages ecosystem and climate change

We now 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 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 *this*" dependencies in Java 8's new parallelism for multi core.

1.1.2 Stream processing

The first programming concept is the idea of *stream processing*. A *stream* is a sequence of data items that are produced one at a time. One practical example is in Unix or Linux where many programs operate by reading data from *stdin* (standard input), operating on it, and then

writing their results on *stdout* (standard output). The Unix command line allows such programs to be linked together with pipes (written |) giving examples such as

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

which, supposing `file1` and `file2` contain words, prints the three words from the files that appear latest in dictionary order (after first translating them to lowercase). You say that `sort` takes a *stream* of lines[2] as input and produces another stream of lines as output (the latter being sorted) as illustrated in figure 1.2. Note that in Unix the programs (`cat`, `tr`, `sort`, and `tail`) are executed concurrently, so that `sort` can process the first few lines before `cat` or `tr` has finished.

Figure 1.2 Unix commands operating on streams

Java 8 adds a Streams API (note the upper-case 'S') in `java.util.stream` based on this idea – `Stream<T>` is a sequence of items of type T. The Streams API has many methods that can be chained to form a complex pipeline just like unix commands were chained above.

The key motivation for this is that Java can now deal with the data at a higher level of abstraction, rather than one item at a time. As a result, it can transparently run your pipeline of Stream operations on several CPU cores on disjoint parts of the input – this is parallelism almost for free. We cover the Java 8 Streams API in detail in Chapter 4.

1.1.3 Passing code to methods with behavior parameterization

The second programming concept required above is the ability to "pass a piece of code to an API". This sounds awfully abstract. In the Unix example, we might want to tell the sort command to use a custom ordering. While the sort command supports a few parameters to perform different pre-defined sorting such as reverse order, these are limited. For example, in traditional Spanish alphabet the word "llama" appears after "lye". What we really 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, we want to tell a `sort` method to compare using Spanish ordering. You could write a method `compareWithSpanishOrdering` to compare two words but, prior to Java 8, you cannot pass this method to another method! Of course you could create a `Comparator` object to pass to the sort method. However, why should you have to create a new

2 Purists will say a "stream of characters," but it's conceptually simpler to think that sort reorders *lines*.

`Comparator` object with all the verbosity it comes when you really want to re-use an existing piece of behavior? Java 8 adds the ability to pass methods (your code) as arguments to other methods. Figure 1.3 illustrates this idea. We will also refer to this conceptually as *behavior parameterization*. Why is this important? The Streams API is built upon the idea of passing code to parameterize the behavior of its operations – just like we passed `compareWithSpanishOrdering` to parameterize the behavior of `sort`.

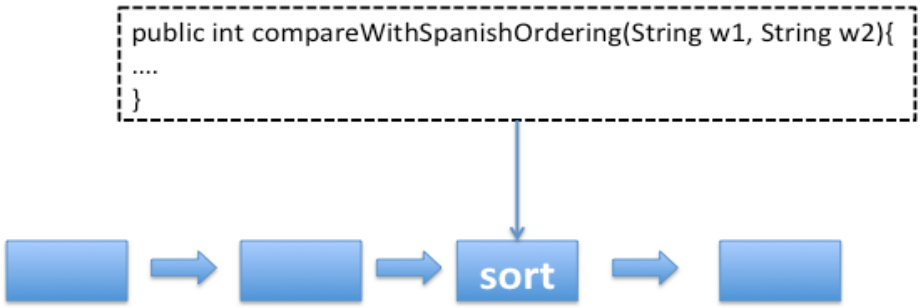


Figure 1.3 Passing a method as argument to sort

We summarize how this works in section 1.2 of this chapter but leave full details to chapter 3. Chapter 10 looks at more advanced things you can do with code-as-a-value beyond passing it to methods.

1.1.4 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 it, you'll love it. You must provide behavior that is safe to execute concurrently on different pieces of the input. Typically this means writing code that does not access shared mutable data to do its job. Sometimes these are referred to as pure functions or side-effect-free functions or stateless functions. 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 1.3 gives a more detailed explanation with a diagram.) You'll find more about this style throughout the book.

Java 8 streams exploit parallelism in an easier way 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 (and using `synchronized` across multiple processing cores is far more expensive than most programmers expect and often counterproductive, 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 is generally described as the paradigm of *functional programming.* 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.

1.1.5 Java needs to evolve

You've seen evolution in Java before. For example, the introduction of generics and your technical lead telling you to use `List<String>` instead of just `List` may have been initially irritating ("Why do we need to change?"), but now we're all increasingly familiar with this style and the benefits it brings (catching more errors at compile time nd making code easier to read, since you now know what something is a list of).

Other changes have made common things easier to express, for example, using a `foreach` 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 more toward functional-programming style in which *what* we 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* we can achieve this (for example, *scan* a data structure *modifying* certain components). Note that classical object-oriented programming and functional programming are not in conflict with each other. It is about trying to get the best from both programming paradigms, so you have more chance of having the right tool for the job! We discuss this in further detail in the next two sections: functional-style programming and Java streams.

A take-away line might be this: to protect your life as a Java programmer, Java has to evolve to keep up. This evolution will be pointless unless the new features are used, so in using Java 8 you're protecting your way of life. On top of that, we have a felling you will 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 enable Java to take over ecological niches currently occupied by other languages (Why? Because Java is well known; lots of programmers; and management decisions like, "the only reason for using rare language XYZ was for the PQR feature, but this is now part of Java, so let's go back to using Java").

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

1.2 Functions in Java

Java 8 adds functions as new forms of value. These facilitate the use of Streams, covered next in section 1.3, which Java 8 provides to exploit parallel programming on multicore 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 implicitly or wrapped inside another library function; object references are *instances* of a class. Examples include `"abc"` (of type `String`), `new Integer(1111)` (of type `Integer`) and the result `new C(7, "xyz")` of explicitly calling a constructor for class `C`. Even arrays are objects. So, what's the problem?

In addition to these values (let's call them first-class citizens of Java following historical tradition), Java has various other concepts that it uses but that, we argue, are second-class Java citizens because they're not values. Methods (and classes) are prime examples of these second-class citizens. Methods are fine when used to define classes, which in turn may be instantiated to produce values, but neither are values. So, does this matter?

Well, it depends whether keeping methods as second-class citizens constrains your thinking (and hence makes it harder to program).

1.2.1 Methods and lambdas as first-class citizens

Experiments in other languages such as Scala and Groovy determined that allowing concepts like methods to be regarded as first-class values made programming easier by adding to the toolset available to the programmers. And once programmers become familiar with a powerful feature, they become reluctant to use languages without it!

So the designers of Java 8 decided to allow methods to be values—to make it easier for us to program and to make it less likely for our manager to choose a different language because Java is not evolving!

The first new Java 8 feature we introduce is the following. 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 is hidden or not. Thankfully there's such a method inside the `File` class called `isHidden()`. It can be viewed as a function that takes a `File` and returns a `boolean`. However, to use it 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(); # A
    }
});
```

#A: filtering hidden files!

Yuck that is horrible! We already have a function `isHidden()` that we could use, why do we have to wrap it up in a verbose `FileFilter` object?

In Java 8 you can rewrite that code as follows:

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

Wow! Isn't that cool? We already have the function `isHidden()` available so we just "pass" it to the `listFiles` method. We will explain later how the mechanics work. However, our code now reads closer to the problem statement. To give you a taste for what's coming, no longer are methods second-class values. The use of `File::isHidden` is a rather special case of a feature called *method references* in Java 8—to be discussed in detail in chapter 3. Given that methods contain code (the executable body of a method), then using methods as values is like passing code around. You'll see a concrete example in the example that follows.

Actually as well as allowing (named) methods to be first-class values, Java 8 allows a richer idea of *functions,* including the idea of *lambdas*3 (or anonymous functions). For example, you can write `(int x)-> x + 1` to mean "the function that, when called with argument x, returns the value x + 1." Chapter 3 explores lambdas in detail.

Programs using such concepts are said to be written in functional-programming style—this phrase essentially just means "writing programs that pass functions around as first-class values."

1.2.2 Passing code: an example

Let's look at an example (in more detail in chapter 2, "Passing code") of how this helps you write programs. All the code for the examples is available on the book's website (http://www.manning.com/urma/), as a download organized by chapter. Suppose you have a class `Apple` with method `getColor`, and an inventory being a list of all `Apples`; then you might wish to select all the green apples and return them in a list. The word *filter* is commonly used to express this concept. You thus might write a method `filterGreenApples`:

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

#A: `result` is a `List`, which accumulates the result; it starts as empty, and then green apples are added one by one.
#B: The highlighted text selects only green apples.

But next, somebody would like the list of heavy apples (say over 150g), and so you write the following (perhaps even using copy and paste):

```
public static List<Apple> filterHeavyApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
```

3 Originally named after the Greek letter λ (lambda). Although the symbol isn't used in Java, its name lives on.

```
        if (apple.getWeight() > 150) { #A
            result.add(apple);
        }
    }
    return result;
}
```

#A: 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 were simply as to what weight range was acceptable, then you could have just passed lower and upper acceptable weights as arguments to filter—perhaps (150,1000) to select heavy apples (over 150g) or (0,80) to select light apples (under 80g)).

But Java 8 makes it possible to pass the code of the whole condition as an argument, thus avoiding code duplication of the `filter` method. You can 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>{ #A
    public boolean test(T t);
}
static List<Apple> filterApples(List<Apple> inventory,
                                Predicate<Apple> p) { #B
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (p.test(apple)) { #C
            result.add(apple);
        }
    }
    return result;
}
```

#A Included for clarity (normally simply imported from `java.util.function`)
#B A method is passed as parameter named p
#C Does the apple match the condition represented by p?

And to use this you simply 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 a method around 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 that takes a value for an argument and returns true or false. In the previous example we could have instead written `Function<Apple,Boolean>`—especially for readers who learned about functions at school but not predicates—but using `Predicate<Apple>` is more standard (and slightly more efficient because it avoids boxing a `boolean`).

1.2.3 From passing methods to lambdas

Passing methods as values is clearly useful, but it's a bit annoying having to write a definition for `isHeavyApple`, `isGreenApple`, and everything else required, 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 just to write

```
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 ||
                                     "brown".equals(a.getColor()) );
```

So 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.

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

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

so programmers don't even have to write methods like `filterApples`.

But, for reasons centered on better exploiting parallelism, they didn't do this. Java 8 instead contains a whole new `Collection`-like API called `Streams`, containing a comprehensive set of operations similar to `filter` that functional programmers may be familiar with (for example, `map`, `reduce`), along with methods to convert between `Collections` and `Streams`. At first this decision was surprising even to programming language experts!

1.3 Java 8: the Streams API and parallelism

Suppose you have a vast amount of data (like Google's index of all web pages); how can you process it? One single CPU wouldn't be able to process this large amount of data. Of course, you probably have a multicore computer on your desk. Ideally, you'd like to share the work among the different CPUs available on your machine to reduce the processing time. In theory, if you have eight CPUs, they should be able to process your data eight times as fast as one CPU because they work in parallel.

> **Multicore**
>
> All new desktop and laptop computers are multicore computers. Instead of a single CPU, they have four, or eight, or more CPUs (cores). The problem is that a classical 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'll need to exploit all the cores in your computer).

The problem is that exploiting parallelism by writing *multithreaded* code (using the `Thread` API from previous versions of Java) is hard. 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[4] properly. This model is harder to think about[5] than a step-by-step sequential model. For example, figure 1.4 shows a possible problem with two `Threads` trying to add a number to a shared variable `sum` if they're not synchronized properly.

4 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.

5 Aha—a source of pressure for the language to evolve!

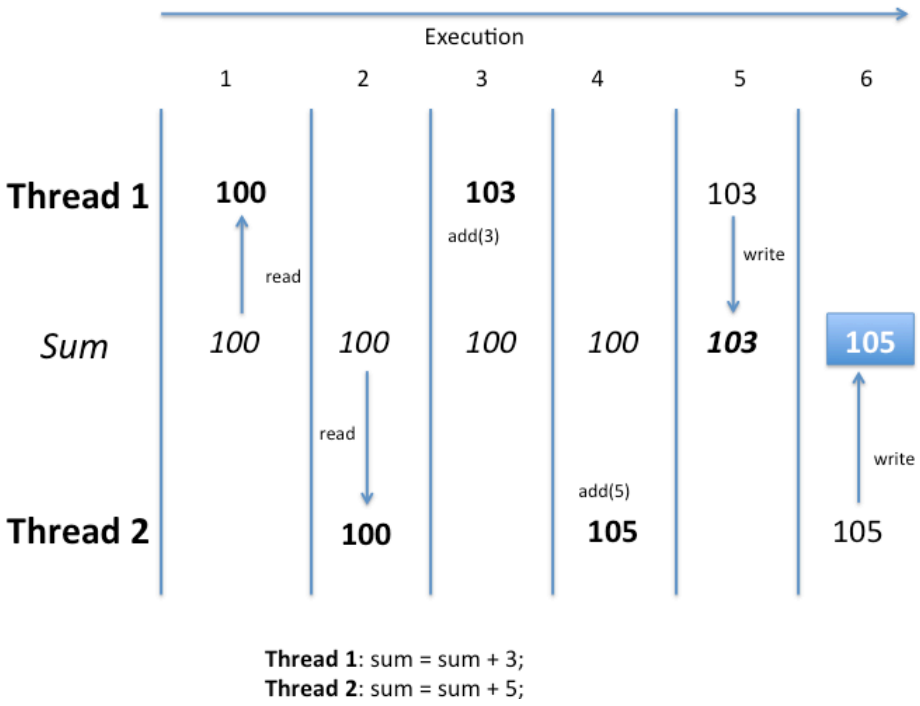


Figure 1.4 A possible problem with two threads trying to add to a shared 'sum' variable. The result is 105 instead of an expected result of 108.

Java 8 addresses this problem by enhancing its library with the `Streams` API (`java.util.stream`), which we explore in detail in chapter 4. The first design motivator is that there are many data processing patterns (similar to `filterApples` in the previous section, or operations familiar from SQL) that occur over and over again and that would benefit from forming part of a library: *filtering* data based on a criteria (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 often be parallelized. For example, 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 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, but using many more than two processors). Figure 1.5 illustrates this idea.

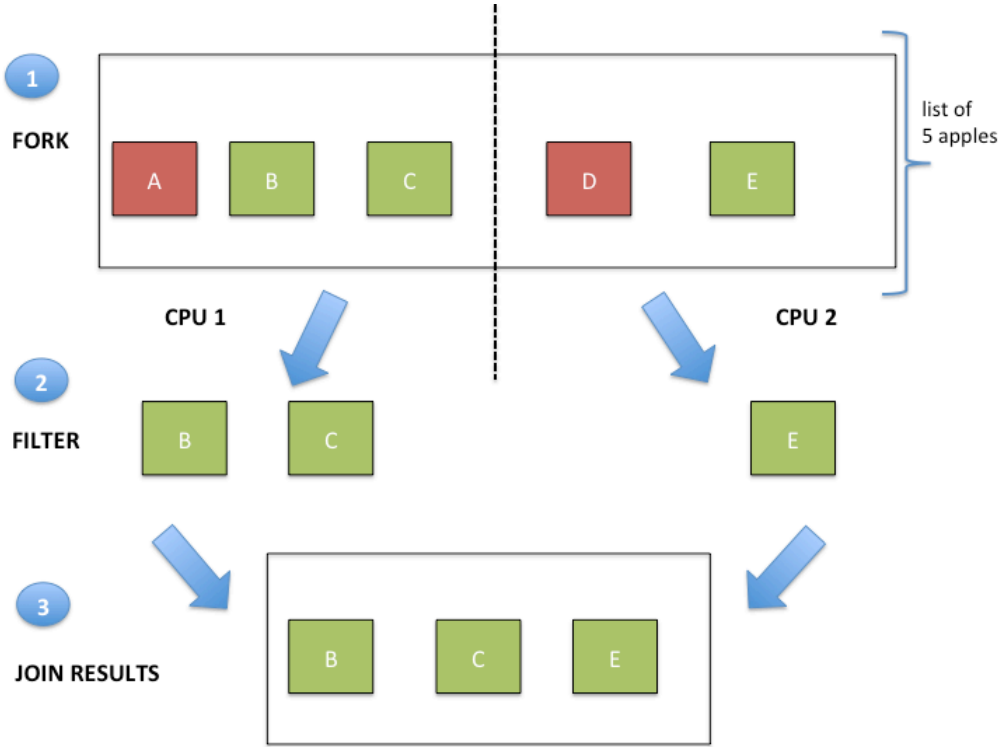


Figure 1.5 Forking `filter` onto two CPUs and joining the result

For now, we'll just say that the new `Streams` API behaves very similarly to Java's existing `Collections` API: both provide access to sequences of data items. However, it's useful for now to keep in mind that collections are mostly about storing and accessing data whereas Streams are mostly about describing computations on aggregates of data. The next section explains the differences in more detail. The key point here is that `Streams` 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` (using `filterApples` on a `List` in the previous section) is to convert it to a `Stream`, process it in parallel, and then convert it back to a `List`, as exemplified here for both the serial and parallel cases. Again we'll just say "parallelism almost for free" and give you a taste of how you can filter heavy apples from a list sequentially or in parallel using `Streams` and a lambda expression:
Sequential processing:

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

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 hard, and all this stuff about synchronized is very error prone. Where's the magic bullet in Java 8? There are actually two magic bullets. First, the library handles partitioning for you. Second, it turns out that 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 (in other words they are stateless). But it turns out this feels quite natural as a coder. Indeed, the secondary meaning of *functional* in *functional programming* often includes "no interaction during execution between components."

Chapter 6 explores parallel data processing in Java 8 and its performance in more detail. In the meantime, we consider why Java 8 added `Streams` and how they differ from `Collections`, which seem to perform a similar role.

1.4 Streams vs. collections

Both the existing Java notion of `Collections` and the new notion of `Streams` provide interfaces to data structures representing a sequenced set of values of the element type. By *sequenced*, we mean that we commonly step through the values in turn, rather than randomly accessing them in any order. So what's the difference?

We'll start with a visual metaphor. Consider a movie stored on a DVD. This is a `Collection` (perhaps of bytes or perhaps of frames—we don't care which here) because it contains the whole data structure. Now consider watching the same video when it is being *streamed* over the internet. This is now a `Stream` (of bytes or frames). The streaming video player only needs to have downloaded a few frames in advance of where the user is watching, so you can start displaying values from the beginning of the `Stream` before the most of the values in a stream have even been computed (consider streaming a live football match). Note particularly that the video player may lack the memory to buffer the whole `Stream` in memory as a `Collection`—and the startup time would be appalling if we had to wait for the final frame to appear before we could start showing the video. We might choose for video-player implementation reasons to *buffer* a part of a `Stream` into a `Collection`, but this is distinct from the conceptual difference.

In coarsest terms, the difference between `Collections` and `Streams` has to do with *when* things are computed. A `Collection` is an in-memory data structure, which holds *all* the values

that the data structure currently has—every element in the Collection has to be computed before it can be added to the Collection. (You can add things to, and remove them from, the Collection, but at each moment in time, every element in the Collection is stored in memory; elements have to be computed before becoming part of the Collection).

By contrast a Stream is a conceptually fixed data structure, in which elements are computed on demand. This gives rise to significant programming benefits. In chapter 5 we show how simple it is to construct a Stream containing all the prime numbers (2,3,5,7,11,...) even though there are an infinite number of them. The idea is that a user will extract only the values they require from a Stream, and these elements are only produced—invisibly to the user—*as* and *when* required. This is form of a producer-consumer relationship. Another view is that a Stream is like a lazily constructed Collection: values are computed when they are solicited by a consumer (in management speak this is demand-driven, or even just-in-time, manufacturing).

In contrast, a Collection is eagerly constructed (supplier-driven: fill your warehouse before you start selling, like a Christmas novelty that has a limited life). Applying this to the primes example, attempting to construct a Collection of all prime numbers would result in a program loop that forever computes a new prime, adding it to the Collection, but of course could never finish making the Collection, so the consumer would never get to see it.

Another example is a browser internet search. Suppose I search for a phrase with many matches in Google or on an e-commerce online shop. Instead of waiting for the whole Collection of results along with their photographs to be downloaded, I get a Stream whose elements are the best 10 or best 20 matches, along with a button to click for the next 10 or 20. When I, as a consumer, click for the next 10, the supplier computes these on demand, before being returned to my browser for display.

Streams and collections philosophically

For readers who like philosophical viewpoints, you can see a Stream as a set of values spread out in time, which repeatedly *appear* at the same point—the argument to a function, passed as a parameter to the Stream-processing function (for example, filter). In contrast, a Collection is a set of values spread out in space (here computer memory), which all exist at a single point of time—and which you access using an iterator to access members inside a foreach loop.

1.4.1 External vs. internal iteration and parallelism

We noted that functional programming style encourages creating new data structures via method or function calls, rather than mutating existing structures—and this agrees with other current pressures (avoiding using synchronized and not sharing mutable data across multiple CPU cores). But using the Collections interface requires iteration to be done by the user (for example, using the enhanced for loop called foreach); this is called *external iteration*. The

Streams library by contrast uses in*ternal iteration*—it does the iteration for you and takes care of storing the resulting stream value somewhere; you merely provide a function saying what's to be done.

Let's use an analogy to understand the difference and benefits of internal iteration. Let's say you are talking to your two-year old daughter Sofia and want her to put her toys away:

You: "Sofia, let's put the toys away. Is there a toy on the ground?"

Sofia: "Yes, the ball."

You: "Okay, put the ball in the box. Is there something else?"

Sofia: "Yes, there's my doll."

You: "Okay, put the doll in the box. Is there something else?"

Sofia: "Yes, there's my book."

You: "Okay, put the book in the box. Is there something else?"

Sofia: "No, nothing else."

You: "Fine, we're finished."

This is exactly what you do everyday with your Java collections. You iterate the collection *externally*, explicitly pulling out and processing the items one by one. It would be far better if you could tell your daughter Sofia just: "Put all the toys that are on the ground inside the box." There are two other reasons why an internal iteration is preferable: first, Sofia could choose to take at the same time the doll with one hand and the ball with the other, and second, she could decide to take the objects closest to the box first and then the others. In the same way using an internal iteration, the processing of items could be transparently done in parallel or in a different order that may be more optimized. These optimizations are very difficult if you iterate the collection externally as you're used to doing in Java (and in imperative programming style in general). This may seem like nit-picking, but it's much of the *raison-d'être* of Java 8's introduction of `Streams`—the internal iteration in the stream library can automatically choose a data representation and implementation of parallelism to match your hardware. By contrast, once you've chosen external iteration by writing `foreach`, then you have essentially committed to self-manage any parallelism. (*Self-managing* in practice means either "one fine day we will parallelize this" or "starting the long and arduous battle involving tasks and `synchronized`".) Hence, Java 8 needed an interface like `Collection` but without iterators, ergo `Streams`! Figure 1.6 illustrates the difference between a Stream (internal iteration) and a Collection (external iteration).

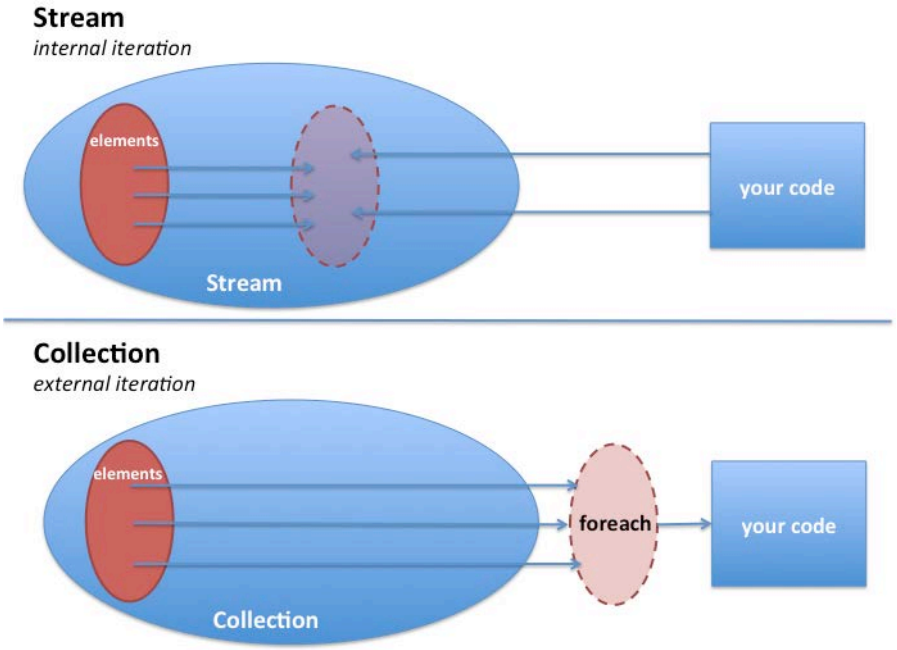


Figure 1.6: Internal vs. external iteration

One of the practical issues the Java 8 developers found was that of evolving existing interfaces. For example, the method `Collections.sort()` really belongs to the `List` interface but was never included. Ideally, we would 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 logistic nightmare! This issue is resolved in Java 8 by *default methods*.

1.5 Default methods

Default methods are added to Java 8 largely to support library designers rather than end programmers. We explain them in detail in chapter 7, but because many programmers won't use them directly, we keep the explanation here short and example based:
In section 1.3, we gave example Java 8 code:

```
List<Apple> inventory = …;
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: an `ArrayList<T>` prior to Java 8 does not have `stream()` or `parallelStream()` methods—and neither does the `Collection<T>` interface that it implements. And without these methods this code will not compile. The simplest solution, which you might employ for your own classes, is to simply add the `stream()` method to the `Collection` interface and add the implementation in the `ArrayList` class.

But doing this would be a nightmare. There are many alternative collection frameworks that implement interfaces from the `Collection` API. Adding a new method to an interface means all concrete classes must provide an implementation for it. Language designers have no control on all existing implementations of `Collections`, so you have a bit of 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! So, 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 uses the new `default` keyword in the interface specification to achieve this (you can think of it as the dual of `abstract`).

For example, in Java 8 you can now 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? So if you have multiple default implementations in several interfaces, that means you have a form of multiple inheritance in Java? Yes, to some extent! We'll show in chapter 8 that there are some restrictions that prevent issues such as the infamous *diamond inheritance problem* in C++.

1.6 Other good ideas from functional programming

The previous two sections have 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, which we present to aid programmers to structure their Java code.

One of these is avoiding `null` by explicit use of more descriptive data types. Indeed, in 2009 Tony Hoare, one of the giants of computer science, wrote:

"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."

In Java 8 there's an `Optional<T>` class that if used consistently can avoid `NullPointer` exceptions. It's a container object that may or not contain a value. `Optional<T>` includes methods to explicitly deal with the case where a value is not absent and as a result you can avoid `NullPointer` exceptions. In other words, it uses the type system to force you to do it right. We discuss `Optional<T>` in detail in chapter 8.

A second idea is that of *pattern matching*. This is used in mathematics, for example:

```
f(0) = 1
f(n) = n*f(n-1) otherwise
```

Whereas in Java you have to write an `if-then-else` or a `switch` statement. As data types become more complex, pattern matching de-clutters code even more. For example, say you want to write a program that does simple arithmetic simplifications. Given data type `Expr` representing an expression, you can write the following code to decompose an expression in the Scala programming language:

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e   // Adding zero
    case BinOp("*", e, Number(1)) => e   // Multiplying by one
    case BinOp("/", e, Number(1)) => e   // Dividing by one
    case _ => expr
}
```

Don't worry about this code for now; we will investigate Scala features such as pattern matching later in the book. For now, you can think of pattern matching as a visitor pattern combined with a switch on steroid that is very useful to decompose a data type.

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 code above it is done using a `match` operation). For example, the visitor pattern is a common pattern used to walk through a family of classes (for example the different components of a car: wheel, engine, chassis…). Pattern matching 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."

Chapter 11 gives 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 12 follows by discussing the language Scala—a language that, like Java, are implemented on top of the JVM and that has evolved quickly to threaten some aspects of Java's niche in the programming language ecosystem.

1.7 Summary

Following are the key concepts you should take away from this chapter:

- The idea of language ecosystem and the consequent evolve-or-wither pressure
- Why multicore processors aren't fully served by existing Java programming practice
- The idea of functions as first-class values, how methods can be passed as functional values, and how anonymous functions (lambdas) are written
- The Java 8 concept of `Streams`, which generalizes many aspects of `Collections` but which allows elements of a stream to be processed in parallel
- The concept of "default method" in an interface to provide a method body if an implementing class chooses not to do so.
- Other interesting ideas from functional programming: dealing with `null` and pattern matching.

2

Passing code with behavior parameterization

This chapter covers

- Adapting to changing requirements
- Abstracting over behavior
- Anonymous classes
- A preview of lambda expressions
- Real-world examples: `Comparator`, `Runnable`, and GUI

A well-known problem in software engineering is that no matter what you do, user requirements will change. For example, imagine an application to help farmers understand their inventory. The farmer might want a functionality to find all green apples in his inventory. But the next day he might ask you, "Actually I also want to find all apples heavier than 150g." Two days later, the farmer comes back and adds, "It would be really nice if I could find all apples that are green *and* heavier than 150g." How can you cope with these changing requirements? Ideally you'd like to minimize engineering efforts. In addition, similar new functionalities ought to be straightforward to implement and maintainable in the long term.

Abstracting over behavior is a pattern that lets you handle frequent requirement changes. In a nutshell, it means taking a block of code and making it available without executing it. This block of code can be called later by other parts of your programs. For example, you could pass the block of code as an argument to another method that will execute it. As a result, the method's behaviour is parameterised based on that block code. For example, if you process a collection you may want to write a method that can do "something" for every element of a list, or do "something else" when you finish the list and do "yet something else" if you encounter an error. This is what *behaviour parameterisation* refers to. Here's an analogy: your roommate

knows how to drive to the supermarket and back home. So you can tell him to buy a list of things such as bread, cheese, and wine. This is equivalent to calling a method goAndBuy with a list of products as argument. But one day you are at the office and you need him to do something he's never done before: pick up a package from the post office. You now need to pass him a list of instructions: "go to the post office, using this reference number, talk to the manager, and pick up the parcel." You could pass him the list of instructions by email, and when he receives it, he can go ahead and follow the instructions. You have now done something a bit more advanced that's equivalent to a method: go, which can take a new behaviour as argument and execute them.

We start the chapter by walking you through an example of how you can evolve your code to be more flexible for changing requirements. Using this knowledge, we show how you use behaviour parameterisation for several real-world examples. For example, you may have already used this pattern using existing classes and interfaces in the Java API to sort a `List`, to filter names of files, or to tell a `Thread` to execute a block of code, or even perform GUI event handling. You'll soon realize that using this pattern is verbose in Java at the moment. Lambda expressions in Java 8 tackle this problem. We show in chapter 3 how to construct lambda expressions, where to use them, and how you can make your code more concise by adopting them.

2.1 Coping with changing requirements

Writing code that can cope with changing requirements is difficult. Let's walk through an example that we will gradually improve, showing some best practices for making your code more flexible. In the context of a farm-inventory application, you have to implement a functionality to filter *green* apples from a list. Sounds easy, right? A first solution might be as follows:

FIRST ATTEMPT: FILTERING GREEN APPLES

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

#A: An accumulator list for apples
#B: Select only green apples

The highlighted line shows the condition required to select green apples. But now the farmer changes his mind and wants to also filter red apples. What can you do? A naïve solution would be to duplicate your method, rename it to `filterRedApples`, and change the `if` condition to match red apples. Nonetheless, this approach doesn't cope well with changes if the farmer

wants multiple different colors: light green, dark red, yellow, and so on. A good principle is: after writing similar code, try to abstract.

SECOND ATTEMPT: ABSTRACTING OVER COLOR

What you could do is add a parameter to your method to parameterize the color and be more flexible to such changes:

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

You can now make the farmer happy and call your method as follows:

```
List<Apple> greenApples = filterApplesByColor(inventory, "green");
List<Apple> redApples = filterApplesByColor(inventory, "red");
…
```

Too easy, right? Let's complicate the example a bit. The farmer comes back to you and says, "It would be really cool to differentiate between light apples and heavy apples. Heavy apples have typically a weight greater than 150g."

Wearing your software engineering hat, you realize in advance that the farmer may want to vary the weight, so you create the following method to cope for various weights through an additional parameter:

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory,
                                               int weight) {
    List<Apple> result = new ArrayList<>();
    For (Apple apple: inventory){
        if ( apple.getWeight() > weight ){
            result.add(apple);
        }
    }
    return result;
}
```

DON'T REPEAT YOURSELF

This is a good solution, but notice how you have to duplicate most of the implementation for traversing the inventory and applying the filtering criteria on each apple. This is somewhat disappointing because it breaks the DRY (Don't repeat yourself) principle of software engineering. What if you want to alter the filter traversing to enhance performance? You now have to modify the implementation of *all* of your methods instead of a single one. This is expensive from an engineering effort perspective.

You could combine the color and weight into one method called `filter`. But then you still need a way to differentiate what attribute you want to filter on. You could add a flag as follows to differentiate between color and weight queries (but never do this! We'll explain why below.):

THIRD ATTEMPT: FILTERING WITH EVERY APPLE ATTRIBUTES WE CAN THINK OF

```
public static List<Apple> filterApples(List<Apple> inventory, String color,
                                       int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if ( (flag && apple.getColor().equals(color)) ||
           (!flag && apple.getWeight() > weight) ){ #A
            result.add(apple);
        }
    }
    return result;
}
```

#A: A really ugly way to select color or weight

You could use it as follows (but it's really ugly!):

```
List<Apple> greenApples = filterApples(inventory, "green", 0, true);
List<Apple> heavyApples = filterApples(inventory, "", 150, false);
…
```

This solution is extremely bad: it doesn't cope well with changing requirements as the behavior of `filterApples` depends on a flag. In addition, the client code looks extremely bad: what does that true or false mean? Moreover, what if the farmer asks you to filter with different attributes of an apple, for example, its size, its shape, its origin, and so on? Furthermore, what if the farmer asks you for more complicated queries that combine attributes, such as green apples that are also heavy. You'd either have multiple duplicated filter methods or one giant, very complex method. So far we've parameterized the `filterApples` method *with values* such as a string, an integer or a boolean. This can be fine for certain well-defined problems. However, in this case what you need is a better way to tell your `filterApples` method the selection criterion for apples. In the next section we describe how to make use of *behavior parameterization* to attain that flexibility.

2.2 Behavior parameterization

You saw in the previous section that you need a better way than adding lots of parameters to cope with changing requirements. Let's step back and find a better level of abstraction. One possible solution is to model our selection criterion (or strategy)6: you are working with apples and returning a `boolean` based on some attributes of Apple (for example, is it green? is

6 For readers familiar with the Gang of Four patterns, this is similar to the strategy design pattern. It is a design pattern that allows you to encapsulate different types of algorithms and choose which one to execute at runtime.

it heavier than 150g?). We call this a *predicate* (that is, a function that returns a boolean). Let's therefore define an interface *to model the selection strategy*:

```
public interface ApplePredicate{
    boolean test (Apple apple);
}
```

You can now declare multiple implementations of `ApplePredicate` to represent different selection strategies, for example (and illustrated in figure 2.1):

```
public class AppleWeightPredicate implements ApplePredicate{ #A
    public boolean test(Apple apple){
        return apple.getWeight() > 150;
    }
}
public class AppleColorPredicate implements ApplePredicate{  #B
    public boolean test(Apple apple){
        return "green".equals(apple.getColor());
    }
}
```

#A: Select only heavy apples
#B: Select only green apples

You can see these filtering strategies as different behaviors for the filter method.

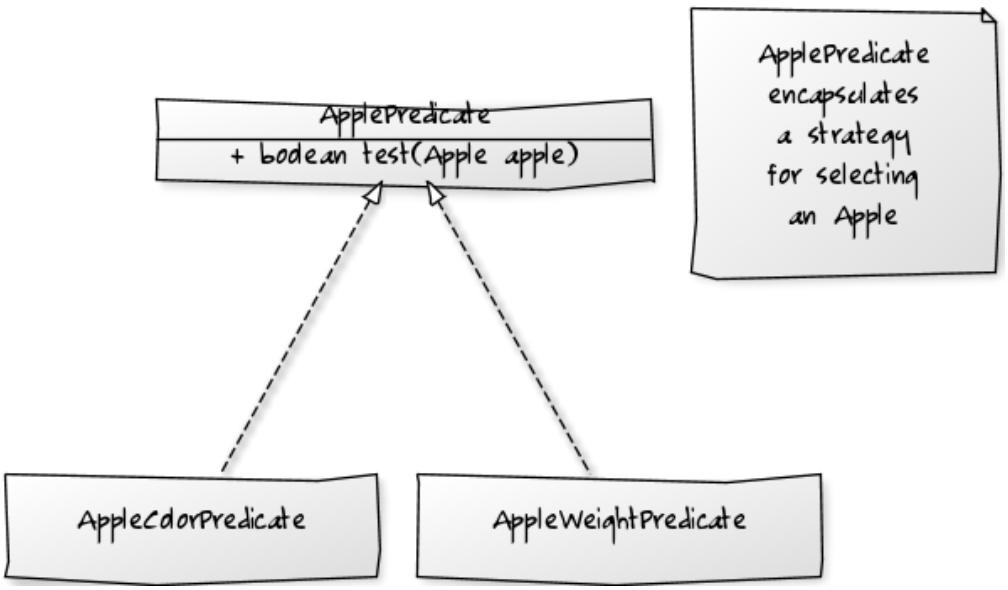


Figure 2.1 Representing different strategies for selecting an Apple.

But how can you actually make use of the different implementations of `ApplePredicate`? You basically need your `filterApples` method to accept `ApplePredicate` objects to test a condition on an Apple. This is what *behavior parameterization* means: the ability to tell a method to take multiple different strategies as parameters and use them internally to accomplish different behaviors.

To achieve this in your example, you add a parameter to the `filterApples` method to take an `ApplePredicate` object. This has a great software engineering benefit: you can separate the logic of iterating the collection inside the `filterApples` method with the behavior you want to apply to each element of the collection (in this case a predicate):

FOURTH ATTEMPT: FILTERING BY AN ABSTRACT CRITERION

```
public static List<Apple> filterApples(List<Apple> inventory,
                                       ApplePredicate p){
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory){
        if(p.test(apple)){ #A
            result.add(apple);
        }
    }
    return result;
}
```

#A: The predicate object encapsulates the condition to test on an apple!

PASSING CODE/BEHAVIOR

It's worth pausing for a moment to have a small celebration. This code is much more flexible than our first attempt, while at the same is easy to read and to use! You can now create different `ApplePredicate` objects and pass them to the `filterApples` method. Free flexibility! For example, if the farmer asks you to find all red apples that are heavier than 150g, all you need to do is create a class that implements the `ApplePredicate` accordingly. You're now flexible for any change of requirements involving the attributes of Apple.

```
public class AppleRedAndHeavyPredicate implements ApplePredicate{
        public boolean test(Apple apple){
                return "red".equals(apple.getColor())
                       && apple.getWeight() > 150;
        }
}

List<Apple> redAndHeavy =
    filter(inventory, new AppleRedAndHeavyPredicate());
```

You've achieved something really cool: the behavior of the `filterApples` method depends on the *code you pass* to it via the `ApplePredicate` object! In other words, you have parameterized the behavior of the `filterApples` method!

Note that in the previous example, the only code that really matters is the implementation of the `test` method, as illustrated in figure 2.2: this is what defines the new behaviors for the `filterApples` method. Unfortunately, because the `filterApples` method can only take objects, you have to wrap that code inside an `ApplePredicate` object. So what you're doing is similar to "passing code" inline, because you're passing a boolean expression through an object that implements the `test` method. You'll see soon that using lambdas, you'll be able to directly pass the expression `"red".equals(a.getColor()) && a.getWeight() > 150` to the `filterApples` method without having to define multiple `ApplePredicate` classes, and as a result removing unnecessary verbosity.

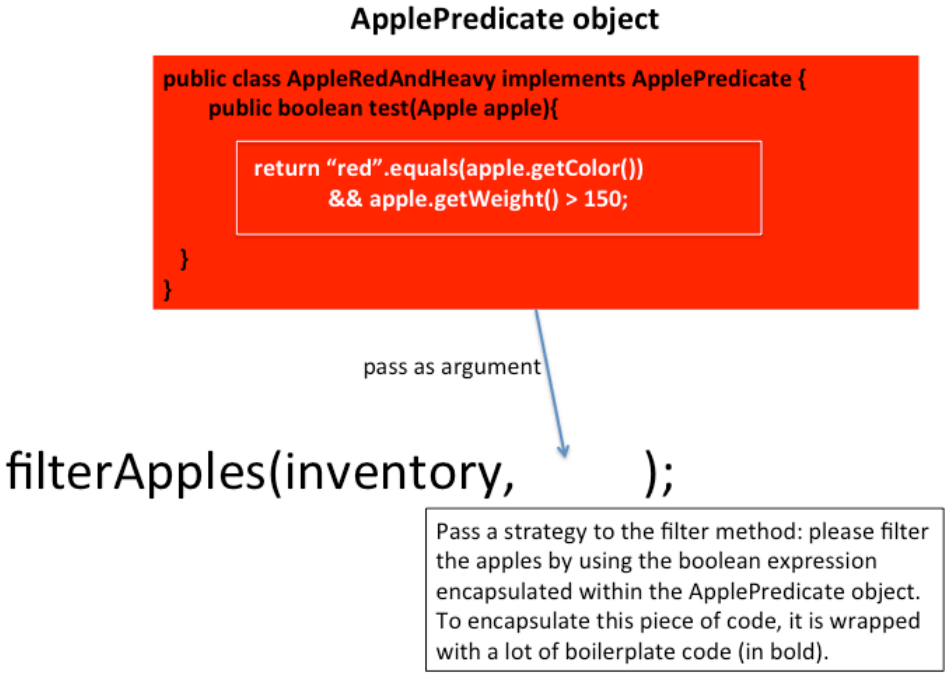


Figure 2.2 Parameterizing the behavior of `filterApples` and passing different filter strategies

MULTIPLE BEHAVIORS, ONE PARAMETER

As we explained earlier, behavior parameterization is great because it enables you to separate the logic of iterating the collection to filter and the behavior to apply on each element of that collection. As a consequence, you can re-use the same method and give it different behaviors to achieve different things, as illustrated in figured 2.3. This is why *behavior parameterization* is a useful concept you should have in your toolset for creating flexible APIs.

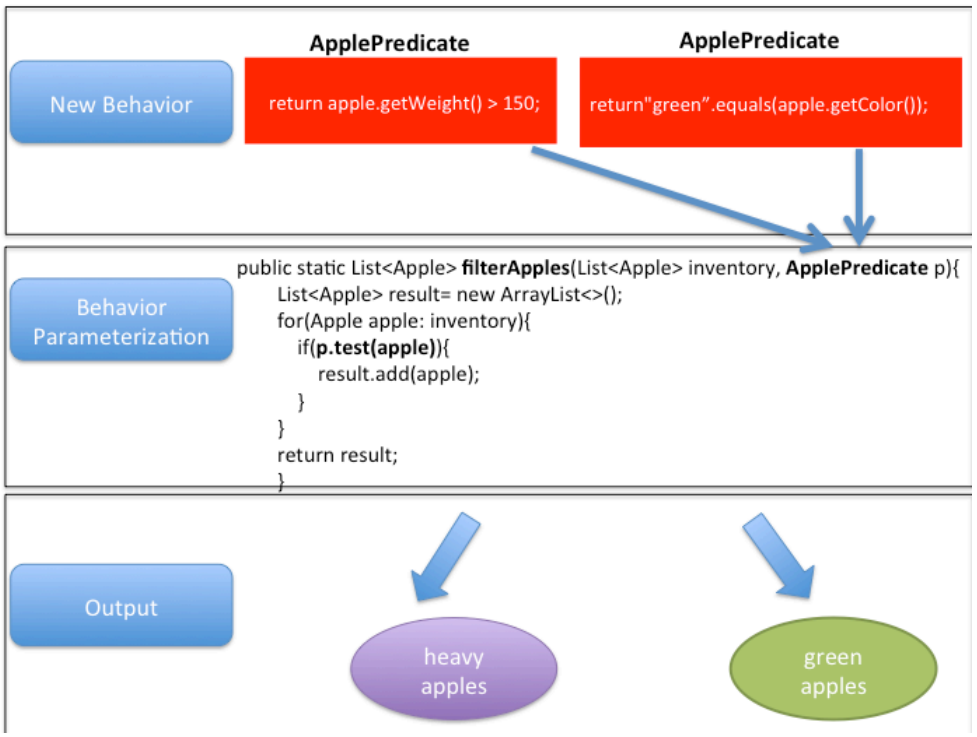


Figure 2.3 Parameterizing the behavior of `filterApples` and passing different filter strategies

To make sure you feel comfortable with the idea of behaviour parameterisation, have a go at Quiz 2.1!

Quiz 2.1: Write a flexible forEachApple

You are to write a `forEachApple` method that takes a `List` of `Apple` and that can be parameterized with multiple behaviors. For example, you should be able to tell your `forEachApple` method to reset all the weights of the apples to 0. In addition, you should be able to tell your `forEach` method to print each `Apple` individually. The solution is similar to the filtering examples we've explored so far. To help you get started we provide a rough skeleton of the `forEach` method:

```
public static void forEachApple(List<Apple> inventory, ???){
    for (Apple apple: inventory) {
        ???
    }
}
```

Answer

First, you need a way to represent a behavior that takes an `Apple` and do something with it without returning a result (that is, `void`). You did something similar when you created an `ApplePredicate` interface.

```
public interface AppleConsumer{
    void accept(Apple a);
}
```

You can now represent multiple behaviors by implementing the `AppleConsumer` interface:

```
public class ApplePrintConsumer implements AppleConsumer{
    public void accept(Apple apple){
        System.out.println(apple);
    }
}
public class AppleResetConsumer implements AppleConsumer{
    public void accept(Apple apple){
        a.setWeight(0);
    }
}
```

Finally, you need to tell your `forEach` method to take `AppleConsumer` objects and use them internally. You do this by adding a parameter to `forEach`:

```
public static forEachApple(List<Apple> inventory, AppleConsumer consumer){
    for(Apple apple: inventory){
        consumer.accept(apple);
    }
}
```

Bingo! You are now able to pass multiple behaviors to your `forEach` method. You do this by instantiating implementations of `AppleConsumer` and giving them as argument to `forEach`:

```
forEachApple(inventory, new AppleResetConsumer());
forEachApple(inventory, new ApplePrintConsumer()); // [0, 0, …]
```

You've seen that we can abstract over behaviour and make our code adapt for requirement changes. However, the process is very verbose. Let's see how to improve that.

2.3 Tackling verbosity

We all know that to encourage the use of a feature or concept, it has to be easy to use. At the moment, when you want to pass new behavior to your `filterApples` method, you are currently forced to declare several classes that implement the `ApplePredicate` interface and then instantiate several `ApplePredicate` objects that you allocate only once, as shown in Listing 2.1. There is a lot of ceremony involved and it is a time-consuming process!

Listing 2.1 Behavior parameterization: filtering apples with predicates

```
public class AppleWeightPredicate implements ApplePredicate{ #A
    public boolean test(Apple apple){
        return apple.getWeight() > 150;
    }
}
public class AppleColorPredicate implements ApplePredicate{  #B
    public boolean test(Apple apple){
        return "green".equals(apple.getColor());
    }
}
import java.util.*;
public class FilteringApples{

    public static void main(String...args){
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));

        List<Apple> heavierThan150g =
            filterApples(inventory, new AppleWeightPredicate());  #C
        List<Apple> greenApples =
            filterApples(inventory, new AppleColorPredicate()); #D

    }
    public static List<Apple> filterApples(List<Apple> inventory,
                                           ApplePredicate p) {
        List<Apple> result = new ArrayList<>();
        for (Apple apple : inventory){
            if (p.test(apple)){
                result.add(apple);
            }
        }
        return result;
    }
}
```

#A: A predicate to select heavy apples
#B: A predicate to select green apples
#C: The result will be a list containing one Apple of 155g
#D: The result will be a list containing two green Apples

This is unnecessary overhead; can you do better? Java has a mechanism called *anonymous classes*, which let you declare and instantiate a class at the same time. They enable you to

improve your code one step further by making it a little more concise. However, they are not entirely satisfactory. We will also show give a short preview of how lambda expressions can make your code more readable before we discuss them in detail in the next chapter.

2.3.1 Anonymous classes

Anonymous classes are like the local classes you are already familiar with in Java (i.e. a class defined in a block) but do not have a name. They allow you to declare and instantiate a class at the same time. In other words, they allow you to create ad hoc implementations. The following code shows how to rewrite the filtering example by creating an object that implements `ApplePredicate` using an anonymous class:

FIFTH ATTEMPT: USING AN ANONYMOUS CLASS

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() { #A
    public boolean test(Apple apple){
        return "red".equals(a.getColor());
    }
});
```

#A: Parameterizing the behavior of the method `filterApples` directly inline!

They're often used in the context of GUI applications to create event-handler objects:

```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

But anonymous classes are still not good enough. First, they tend to be very bulky because they take a lot of space, as shown in the highlighted code in the following example:

```
List<Apple> result = filterApples(inventory, new ApplePredicate() { #A
    public boolean test(Apple a){
        return "red".equals(a.getColor());
    }
});
```

#A: Lots of boilerplate code

```
button.setOnAction(new EventHandler<ActionEvent>() { #A
    public void handle(ActionEvent event) {
        System.out.println("Woooo a click!!");
    }
});
```

#A: Lots of boilerplate code

Second, many programmers find them confusing to use. For example, Quiz 2.2 shows a classic Java puzzler that catches most programmers off guard!

Quiz 2.2: Anonymous class puzzler

What will the output be when this code is executed?

a) 4

b) 5

c) 6

d) 42

```
public class MeaningOfThis
{
    public final int value = 4;
    public void doIt()
    {
        int value = 6;
        Runnable r = new Runnable(){
                public final int value = 5;
                public void run(){
                    int value = 10;
                    System.out.println(this.value);
                }
            };
            r.run();
        }
        public static void main(String...args)
        {
            MeaningOfThis m = new MeaningOfThis();
            m.doIt();                                          # A
        }
}
```

 # A: What's the output of this line?

Answer:

The answer is `5`, because `this` refers to the enclosing `Runnable` not the enclosing class `MeaningOfThis`.

Verbosity in general is bad, because it discourages the use of a language feature because it takes a long time to write and maintain verbose code and is also not pleasant to read! Good code should be easy to comprehend at a glance. Even though anonymous classes somewhat tackle the verbosity associated with declaring multiple concrete classes for an interface, in the context of just passing a piece of code, you still have to create an object and explicitly implement a method to define a new behavior (for example the method test for `Predicate` or handle for `EventHandler`).

Ideally we'd like to encourage programmers to use the behaviour parameterisation pattern, because as you've just seen, it makes your code more adaptive to requirement

changes. In chapter 3 you'll see that the Java 8 language designers solved this problem by introducing lambda expressions: a more concise way to pass code. Enough suspense, let's have a short preview of how lambda expressions can help you in your mission towards clean code!

2.3.2 Preview of lambda expressions

Our previous code can be rewritten as follows in Java 8 using a lambda expression:

SIXTH ATTEMPT: USING A LAMBDA EXPRESSION

```
List<Apple> result =
  filterApples(inventory, (Apple apple) -> "red".equals(apple.getColor()));
```

You have to admit this code looks a lot cleaner than our previous attempts! It's great because it is starting to look a lot closer to the problem statement. We've now tackled the verbosity issue. Figure 2.4 summarises our journey so far.

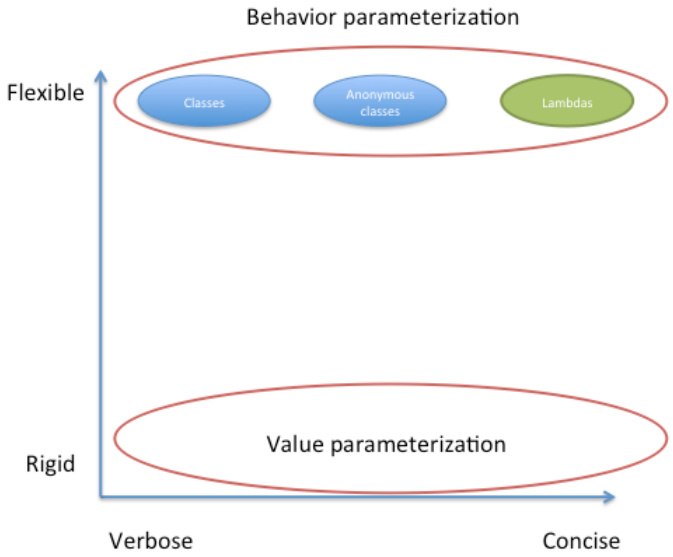


Figure 2.4: Behavior parameterization vs. value parameterization

For the avid readers, there's one more step that we can do in our journey towards abstraction. At the moment our `filterApples` method only works for Apple. However, we can also abstract on the list type to go beyond the problem domain we are thinking of right now:

SEVENTH ATTEMPT: ABSTRACTING OVER LIST TYPE

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

```
public static <T> List<T> filter(List<T> list, Predicate<T> p){
    List<T> result = new ArrayList<>();
    for(T e: list){
        if(p.test(e)){
            result.add(e);
        }
    }
    return result;
}
```

We can now use the method filter with a list of bananas, oranges, integers or strings! For example, using lambda expressions:

```
List<Apple> redApples =
  filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));

List<String> evenNumbers =
  filter(numbers, (Integer i) -> i % 2 == 0);
```

Isn't it cool? We have managed to find the sweet spot between flexibility and conciseness, which wasn't possible prior to Java 8!

2.4 Real-world examples

You've now seen that behavior parameterization is a useful pattern to easily adapt to changing requirements. In fact, this pattern lets you encapsulate an arbitrary strategy and parameterize the behavior of methods, by using the strategies you implement (for example, different predicates for an `Apple`). We mentioned earlier that this approach is similar to the strategy design pattern from the Gang of Four. You may have already used this pattern in practice. In fact, many methods existing in the Java API can be parameterized with different behaviors. These methods are often used together with anonymous classes. We'll show you three examples, which should solidify the idea of passing code for you: sorting with a `Comparator`, executing a block of code with `Runnable`, and GUI event handling.

2.4.1 Sorting with a comparator

Sorting a collection is a frequent programming task. For example, say your farmer wants you to sort the inventory of apples based on their weight. Or perhaps he changes his mind and wants you to sort the apples by color. Sounds familiar? Yes, you need a way to represent and use different sorting behaviors to easily adapt to changing requirements!

In Java 8, a `List` comes with a `sort` method (you could also use `Collections.sort`). The behavior of `sort` can be parameterized using a `java.util.Comparator` object, which has the following interface:

```
// java.util.Comparator
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

You can therefore create different behaviors for the `sort` method by creating an ad hoc implementation of `Comparator`. For example, you can use it to sort the inventory by increasing weight using an anonymous class:

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

If the farmer changes his mind about how to sort apples, you can create an ad hoc `Comparator` to match new the requirement and pass it to the sort method! The internal details of how to sort are abstracted away. With a lambda expression it would look like this:

```
inventory.sort(
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

2.4.2 Executing a block of code with Runnable

Threads are like a lightweight process: they execute a block of code on their own. But how can you tell a thread what block of code to run? Several threads may run different code. What you need is a way to represent a piece of code to be executed later. In Java, you can use the `Runnable` interface to represent a block of code to be executed (note that the code will return no result (that is, `void`)):

```
// java.lang.Runnable
public interface Runnable{
    public void run();
}
```

You can use this interface to create threads with different behaviors as follows:

```
Thread t = new Thread(new Runnable() {
    public void run(){
        System.out.println("Hello world");
    }
});
```

With a lambda expression it would like this:

```
Thread t = new Thread(() -> System.out.println("Hello world"));
```

2.4.3 GUI event handling

A typical pattern in GUI programming is to perform a response given a certain event such as clicking or hovering over text. For example, if the user clicks a Send button, you may wish to display a popup or perhaps log the action in a file. Again, you need a way to cope for changes:

you should be able to perform any response. In JavaFX (a modern UI platform for Java) you can use an `EventHandler` to represent a response to an event by passing it to `setOnAction`:

```
Button button = new Button("Send");
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        label.setText("Sent!!");
    }
});
```

Here, the behavior of the `setOnAction` method is parameterized with `EventHandler` objects. With a lambda expression it would look like this:

```
button.setOnAction((ActionEvent event) -> label.setText("Sent!!"));
```

2.5 Summary

Following are the key concepts you should take away from this chapter:

- *Behavior parameterization* is the ability for a method to take multiple different strategies as parameter and use them internally to accomplish different behaviors.
- *Behavior parameterization* lets you make your code more adaptive to changing requirements and saves engineering efforts in the future.
- *Passing code* is a way to give new behaviors as argument to a method. But it is currently verbose in Java. *Anonymous classes* help a bit to get rid of the verbosity associated with declaring multiple concrete classes for an interface that are needed only once.
- The Java API contains many methods that can be parameterized with behavior, which include sorting, threads, and GUI handling.

3

Lambda expressions

This chapter covers

- Lambdas and where to use them
- The Execute Around pattern
- Functional interfaces, type inference
- Method references
- Composing lambdas

In the previous chapter, you saw that passing code with behavior parameterization is useful for coping with frequent requirement changes in your code. It lets you pass around a block of code that represents a behavior *so it can be executed later*. You can decide to run a piece of code only when a certain event happens (e.g. a click on a button) or at certain points in an algorithm (e.g. a predicate “only apples heavier than 150g” in the filtering algorithm or the customized comparison operation in sorting). In general, using this concept you can write code that is more flexible and reusable.

But you saw that using anonymous classes to represent different behaviors is unsatisfying: it is verbose, which doesn’t encourage programmers to use behavior parameterization in practice. In this chapter, we teach you about a new feature in Java 8 that tackles this problem: lambda expressions, which let you represent a behaviour or “pass code” in a concise way. For now you can think of *lambda expressions* as anonymous functions, basically methods without declared names, but which can also be passed around like an anonymous class.

We show how to construct them, where to use them, and how you can make your code more concise by using them. We also explain some new goodies such as type inference and new important interfaces available in the Java 8 API. Finally, we introduce method references, a useful new feature that goes hand in hand with lambda expressions.

This chapter is organized in such a way as to teach you step by step how to write more concise and flexible code. At the end of this chapter we will bring all the concepts taught into a concrete example: we take the sorting example seen in chapter 2 and gradually improve it using lambdas expressions and method references to make it more concise and readable. This chapter is important because you will use lambdas extensively in the next chapter about streams.

3.1 In a nutshell

A *lambda expression* can be understood as a kind of anonymous function that can be passed around: it doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown. You saw in the previous chapter that passing code is currently tedious and verbose in Java. Well, good news! Lambdas fix this problem: they let you pass code in a concise way. Thus, you can cope with changing requirements by using a behavior, represented by a lambda, as a parameter to a method. Lambdas do not let you do anything that you couldn't prior to Java 8. However, you no longer have to write clumsy code using anonymous classes to benefit from behavior parameterization! If you are wondering where the term *lambda* comes from, it originates from a system developed in academia called *lambda calculus*, which is used to describe computations. That's one big definition; let's break it down:

- *Anonymous*—We say *anonymous* because it doesn't have an explicit name like a method would normally have: less to write and think about!
- *Function*—We say *function* because a lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exception that can be thrown.
- *Passed around*—A lambda expression can be passed as argument to a method or stored in a variable.
- *Concise*—You don't need to write a lot of boilerplate like you do for anonymous classes.

Why should you care? Using lambda expressions will encourage you to adopt the style of *behavior parameterization* that we described in the previous chapter, because lambdas let you pass code in a concise way compared to anonymous classes. As a result, your code will be clearer and more flexible. You'll see in the next chapter that lambdas are used all over the place with a new API called Streams that lets you process collections with SQL-like operations.

Using a lambda expression we can create a custom `Comparator` object in a more concise way.

BEFORE:

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

AFTER (WITH LAMBDA EXPRESSIONS):

```
Comparator<Apple> byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

You must admit that the code looks clearer! Don't worry if all the parts of the lambda expression don't make sense yet; we'll explain all the pieces soon. For now, note that we're literally passing only the code that's really needed to compare two apples using their weight. It looks like we're just passing the body of the method `compare`. You'll learn soon that you can simplify your code even more. We explain in the next section exactly where and how you can use lambda expressions.

The lambda we just showed you has three parts, as shown in figure 3.1.

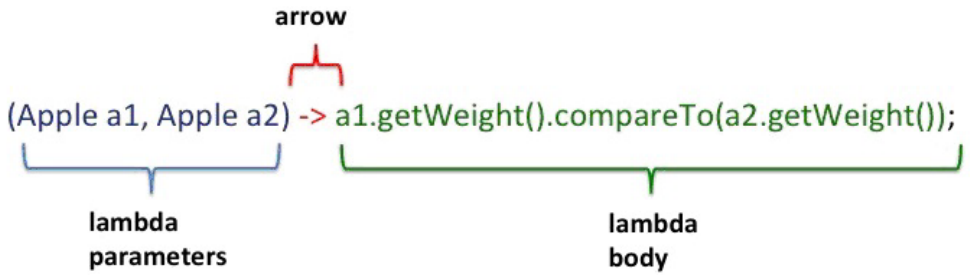


Figure 3.1 A lambda expression is composed of parameters, an arrow, and a body.

- *A list of parameters*—In this case it mirrors the parameters of the `compare` method of a `Comparator`—two Apples.
- *An arrow*—`->` to separate the list of parameters from the body of the lambda.
- *The body of the lambda*—Compare two apples using their weights.

To illustrate further, here are five examples of valid lambda expressions in Java 8:

```
1. (String s) -> s.length()
2. (Apple a) -> a.getWeight() > 150
3. (int x, int y) -> { System.out.println(x+y); }
4. () -> 42
5. (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
```

- The first lambda expression has one parameter of type `String` and returns an `int`. The lambda doesn't have a `return` statement here because the `return` is implied.
- The second lambda expression has one parameter of type `Apple` and returns a `boolean` (that is, whether the apple is heavier than 150g).
- The third lambda expression has two parameters of type `int` with no return (`void`

return).

- The fourth lambda expression has no parameter and returns an `int`.
- The fifth lambda expression has two parameters of type `Apple` and returns an `int`: the comparison of the weight of the two Apples.

This syntax was chosen by the Java language designers because it was well received in other languages such as C# and Scala, which have a similar feature. The basic syntax of a lambda is either:

```
(parameters) -> expression
```

or (note the curly braces for statements)

```
(parameters) -> { statements; }
```

As you can see, lambda expressions follow a simple syntax. Working through Quiz 3.1 below, should let you know if you understand the pattern.

Quiz 3.1: Lambda syntax

Based on the syntax rules just shown, which of the following are not valid lambda expressions?

```
() -> {}
() -> "Raoul"
() -> {return "Mario";}
(Integer i) -> return "Alan" + i;
(String s) -> {"Iron Man";}
```

Answer: Only 4 and 5 are invalid lambdas.

1) A lambda that has no parameters and returns `void`. It's similar to a method with an empty body: `public void run() { }`

2) A lambda that has no parameters and returns a `String` as an expression.

3) A lambda that has no parameter and returns a `String` (using an explicit return statement).

4) `return` is a control flow statement. To make this lambda valid, curly brackets are required as follows: `(Integer i) -> {return "Alan" + i;}`

5) "Iron Man" is an expression, not a statement. To make this lambda valid, you can remove the curly brackets and semicolon as follows: `(String s) -> "Iron Man"` or if you prefer you can use an explicit return statement as follows: `(String s) -> {return "Iron Man";}`

Table 3.1 provides a list of examples lambdas with examples of use cases:

Table 3.1 Examples of lambdas

Use case	Example of lambda
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`
Consuming from an object	`(Apple a) -> System.out.println(a.getWeight())`
Select/extract from an object	`(String s) -> s.length()`
Combine two values	`(int a, int b) -> a * b`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`

3.2 Where and how to use lambdas

You may now be wondering where you're allowed to use lambda expressions. In the previous example, we assigned a lambda to a variable of type `Comparator<Apple>`. You could also use another lambda with the `filter` method we implemented in the previous chapter:

```
List<Apple> greenApples =
        filter(inventory, (Apple a) -> "green".equals(a.getColor()));
```

So where exactly can you use lambdas? To answer this question we need to explain what the type of a lambda expression is, because Java lets you pass expressions only where a type is expected (for example, `String` and `List` are types). The type of a lambda expression is essentially a *functional interface*.

3.2.1 Functional interface

Remember the interface `Predicate<T>` we created earlier so we can parameterize the behavior of the `filter` method? It's a functional interface! Why? Because, `Predicate` specifies only one abstract method:

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

In a nutshell, a *functional interface* is an interface that specifies exactly one method. To be slightly more accurate, we should say "that specifies exactly one *abstract* method" (that is, a method that has no body; it's the signature of a method). You already know several other functional interfaces in the Java API such as `Comparator` and `Runnable`, which we explored in chapter 2:

```
// java.util.Comparator
public interface Comparator<T> {
    public int compare(T o1, T o2);
```

```
}

// java.lang.Runnable
public interface Runnable{
    public void run();
}

// java.awt.event.ActionListener
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}

// java.util.concurrent.Callable
public interface Callable<V>{
    public V call();
}

// java.security.PrivilegedAction
public interface PrivilegedAction<V>{
    public T run();
}
```

As a side note, you'll see in chapter 8 that interfaces can now also have *default methods* (that is, a method with a body that provides some default implementation for a method in case it isn't implemented by a class). An interface is still a functional interface if it has many default methods as long as it specifies *only one abstract method*. To check your understanding, Quiz 3.2 should let you know if you understand the concept of a functional interface.

Quiz 3.2: Functional interface

Which of these interfaces are functional interfaces?

```
public interface Adder{
    public int add(int a, int b);
}
public interface SmartAdder extends Adder{
    public int add(double a, double b);
}
public interface Nothing{
}
```

Answer: Only `Adder` is a functional interface.

`SmartAdder` is not a functional interface because it specifies two abstract methods called `add` (one is inherited from `Adder`).

`Nothing` is not a functional interface because it declares no abstract method at all.

So what can you do with functional interfaces? Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and *treat the whole expression as an instance of a functional interface*. You can achieve the same thing with an anonymous inner class, although it is clumsier: you provide an implementation and instantiate it directly inline. The following code is valid because `Runnable` is a functional interface defining only one abstract method, `run`:

```
Runnable r1 = () -> System.out.println("Hello World 1"); #A

Runnable r2 = new Runnable(){ #B
    public void run(){
        System.out.println("Hello World 2");
    }
}

public void process(Runnable r){
    r.run();
}
process(r1); #C
process(r2); #D
process(() -> System.out.println("Hello World 3")); #E
```

#A: Using a lambda
#B: Using an anonymous class
#C: Prints "Hello World 1"
#D: Prints "Hello World 2"
#D: Prints "Hello World 3" with a lambda passed directly!

3.2.2 Function descriptor

The signature of the abstract method of the functional interface essentially describes the signature of the lambda expression. We call this abstract method a *function descriptor*. For example, the `Runnable` interface can be viewed as the signature of a function that accepts nothing and returns nothing (`void`) because it has only one abstract method called `run`, which accepts nothing and returns nothing (`void`)7.

It doesn't matter what the single method or interface is called. Other languages that have the concept of "function types" might represent this signature with the notation `() -> void`: a function with an empty list of parameters returning `void`.

We'll use this notation throughout this chapter to describe signatures of lambdas and functional interfaces. For example, `(Apple, Apple) -> int` denotes a function taking two `Apples` as parameters and returning an `int`.

We detail how the compiler checks whether a lambda is valid in given context in section 3.5. For now, it suffices to understand that a lambda expression can be assigned to a variable or passed to a method expecting a functional interface as argument, provided the lambda

7 Some languages such as Scala provide explicit type annotations in its type system to describe the type of a function (called function types). Java re-uses existing nominal types provided by functional interfaces and maps them into a form of function types behind the scene.

expression has the same signature as the abstract method of the functional interface. For instance, you could pass a lambda directly to the process method as follows:

```
public void process(Runnable r){
    r.run();
}

process(() -> System.out.println(“This is awesome!!”));
```

This code when executed will print “This is awesome.” The lambda expression `() -> System.out.println("This is awesome")` takes no parameters and returns void. This is exactly the signature of the run method defined in the Runnable interface.

You may be wondering, “Why can we only pass a lambda where a functional interface is expected?” The language designers considered alternative approaches such as adding “function types” to Java. However, they chose this way because it fits naturally without increasing complexity. In addition, most Java programmers are already familiar with the concept of an interface with a single abstract method (e.g. in Swing).

Quiz 3.3: Where can you use lambdas?

What are valid uses of lambda expressions?

1.

```
execute(() -> {});
public void execute(Runnable r){
  r.run();
}
```

2.

```
public Callable<String> fetch() {
  return () -> "Tricky example ;-)";
}
```

3.

```
Predicate<Apple> p = (Apple a) -> a.getWeight();
```

Answer: Only 1 and 2 are valid.

1 is valid because the lambda `() -> {}` has the following signature: `() -> void`, which matches the signature of the abstract method `run()` defined in `Runnable`. Note that running this code will do nothing because the body of the lambda is empty!

2 is valid. Indeed the return type of the method `fetch` is `Callable<String>`. `Callable<String>` essentially defines a method with the signature: `() -> String` when `T` is replaced with `String`. Because the lambda `() -> "Tricky example ;-) "` has the signature `() -> String`, the lambda can be used in this context.

3 is invalid because the lambda expression `(Apple a) -> a.getWeight()` has the signature `(Apple) -> int`, which is different than the signature of the method test defined in `Predicate<Apple>: (Apple) -> boolean`.

What about @FunctionalInterface?

If you explore the new Java API, you'll notice that functional interfaces are annotated with `@FunctionalInterface` (we show an extensive list in section 3.4, where we explore functional interfaces in depth). This annotation is used to indicate the interface is intended to be a functional interface. The compiler will return a meaningful error if you define an interface using the `@FunctionalInterface` annotation and it isn't a functional interface. For example, an error message could be: "multiple non-overriding abstract methods found in interface Foo" to indicate that more than one abstract methods are available.

3.3 Putting lambdas in practice: the "execute around" pattern

Let's look at an example of how *lambdas,* together with *behavior parameterization,* can be used in practice to improve your code. A recurrent pattern in resource processing (for example, dealing with files or databases) is to open a resource, do some processing on it, and then close the resource. The setup and cleanup phases are always similar and appear around the important code doing the processing. This is called the *execute around* pattern as illustrated in figure 3.2. For example, in the following code, the highlighted lines show the boilerplate code required to read one line from a file (note also we use Java 7's try-with-resources, which already simplifies the code, because we don't have to close the resource explicitly):

```
public static String processFile() throws IOException {
    try (BufferedReader br =
             new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();              #A
    }
}
```

#A: This is the line that does useful work.

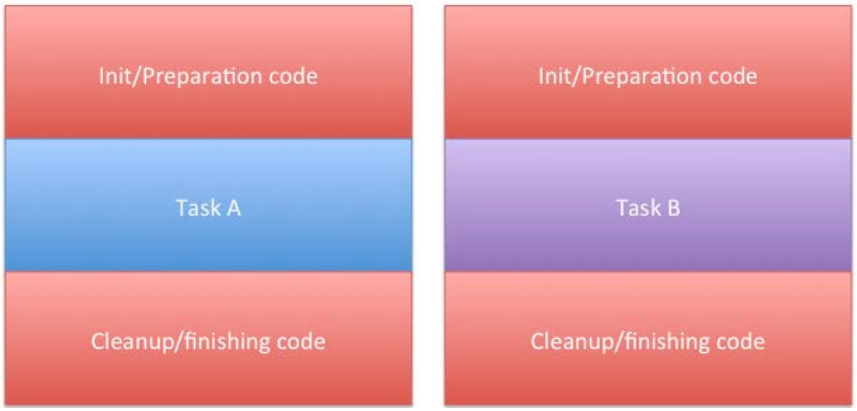


Figure 3.2 Task A and B are surrounded by the same redudant code responsible for preparation/cleanup.

3.3.1 Step 1: Remember behavior parameterization?

This current code is limited. You can read only the first line of the file. What if you'd like to return the first two lines instead, or even count the most frequent word and return it? Ideally, you'd like to reuse the code doing setup and cleanup and tell the `processFile` method to perform different actions on the file. Does this sound familiar?! Yes, you need to parameterize the behavior of `processFile`. You need a way to pass behavior to the `processFile` so it can execute different behaviors using a `BufferedReader`.

Passing behavior is exactly what lambdas are for. So what should the new `processFile` method look like? You basically need a lambda that takes a `BufferedReader` and returns a `String`. For example, here's how to print two lines of a `BufferedReader`:

```
String result = processFile((BufferedReader br) ->
                            br.readLine() + br.readLine());
```

3.3.2 Step 2: Using a functional interface to pass behaviors

We explained earlier that lambdas can only be used in the context of a functional interface. So you need to create one that matches the signature `(BufferedReader b) -> String`. Let's call this interface `BufferedReaderProcessor`:

```
@FunctionalInterface
public interface BufferedReaderProcessor {
  public String process(BufferedReader b) throws IOException;
}
```

You can now use this interface as the argument to your new `processFile` method:

```
public static String processFile(BufferedReaderProcessor p) throws
IOException {
   …
}
```

3.3.3 Step 3: Actually executing a behavior!

So any lambdas of the form `(BufferedReader b) -> String` can be passed as argument, because they match the signature of the `process` method defined in the `BufferedReaderProcessor` interface. You now only need a way to actually execute the code represented by the lambda inside the body of `processFile`. As you remember, lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline, and they *treat the whole expression as an instance of a functional interface.* You can therefore call the method `process` on the resulting `BufferedReaderProcessor` object inside the `processFile` body to perform the processing:

```
public static String processFile(BufferedReaderProcessor p) throws
IOException {
    try (BufferedReader br =
                  new BufferedReader(new FileReader("data.txt"))) {
        return p.process(br);
    }
}
```

3.3.4 Step 4: Passing lambdas

You can now reuse the `processFile` method and process files in different ways by passing different lambdas:

PROCESSING ONE LINE

```
String oneLine =
  processFile((BufferedReader br) -> br.readLine())
```

PROCESSING TWO LINES

```
String twoLines =
  processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

Figure 3.3 summarizes the four steps taken to make the `processFile` method more flexible.

Figure 3.3 Four-step process to apply the execute around pattern

So far we showed you how you can make use of functional interfaces to pass lambdas. But you had to define your own interfaces. In the next section we explore new interfaces that were added to Java 8 that you can reuse to pass multiple different lambdas!

3.4 Using functional interfaces

As you learned in section 3.2.1, a functional interface specifies exactly one abstract method. Functional interfaces are useful because the signature of the abstract method can describe the signature of a lambda expression. The signature of the abstract method of a functional interface is called a *function descriptor*. So, in order to use different lambda expressions, you need a set of functional interfaces that can describe common function descriptors. There are several functional interfaces available in the Java API such as `Comparable`, `Runnable`, and `Callable`.

But the Java language designers for Java 8 have helped us by introducing several new functional interfaces inside the `java.util.function` package. We describe the interfaces `Predicate`, `Consumer`, and `Function` next, and a more complete list is available in table 3.2 at the end of this section.

3.4.1 Predicate

The `java.util.function.Predicate<T>` interface defines an abstract method named `test`, which accepts an object of generic type `T` and returns a `boolean`. It is exactly the same one that you created yourself earlier but is available out of the box! You might want to use this interface when you need to represent a boolean expression that uses an object of type `T`. For example, you can define a lambda that accepts `String` objects:

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

public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for(T s: list){
        if(p.test(s)){
            results.add(s);
        }
    }
    return results;
}

Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpt = filter(listOfStrings, nonEmptyStringPredicate);
```

If you look up the Javadoc specification of the `Predicate` interface you may notice additional methods such as and and or. Don't worry about them for now. We come back to these in section 3.8.

3.4.2 Consumer

The `java.util.function.Consumer<T>` interface defines an abstract method named `accept`, which takes an object of generic type `T` and returns no result (`void`). You might use this interface when you need to access an object of type `T` and perform some operations on it. For example, you can use it to create a method `forEach`, which takes a list of `Integer` and applies an operation on each element of that list in order. In the following example we use this `forEach` method combined with a lambda to print all the elements of the list:

```
public interface Consumer<T>{
    public void accept(T t);
}

public static <T> void forEach(List<T> list, Consumer<T> c){
    for(T i: list){
        c.accept(i);
    }
}

forEach(
         Arrays.asList(1,2,3,4,5),
        (Integer i) -> System.out.println(i) #A
       );
```

#A: The lambda is the implementation of the `accept` method from `Consumer`

3.4.3 Function

The `java.util.function.Function<T, R>` interface defines an abstract method named `apply`, which takes an object of generic type `T` as input and returns an object of generic type `R`. You might use this interface when you need to define a lambda that can extract information from the input object (for example, extracting the weight of an apple) or transform the input (for example, a string to its length). In the code that follows we show how you can use it to create a method `map` to transform a list of `String` into a list of `Integer` containing the length of each `String`:

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

public static <T, R> List<R> map(List<T> list,
                                 Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T s: list){
        result.add(f.apply(s));
    }
    return result;
}

// [7, 2, 6]
List<String> l = map(
                        Arrays.asList("lambdas","in","action"),
                        (String s) -> s.length() #A
                );
```

#A: The lambda is the implementation for the `apply` method of `Function`

PRIMITIVE SPECIALIZATIONS

We described three functional interfaces that are generic: `Predicate<T>`, `Consumer<T>`, and `Function<T, R>`. There are also functional interfaces that are specialized with certain types.

To refresh a little: every Java type is either a reference type (for example, `Byte`, `Integer`, `Object`, `List`) or a primitive type (for example, `int`, `double`, `byte`, `char`). But generic parameters (for example the `T` in `Consumer<T>`) can only be bound to reference types. This is due to how generics are internally implemented. Some other languages such as C# don't have this restriction. Other languages such as Scala only have reference types. As a result, in Java there's a mechanism to convert a primitive type into a corresponding reference type. This mechanism is called *boxing*. The dual (that is, converting a reference type into a corresponding primitive type) is called *unboxing*. Java has also a mechanism called *autoboxing* to facilitate the task for programmers: boxing and unboxing operations are done automatically. For example, this is why the following code is valid (an `int` gets boxed to an `Integer`):

```
List<Integer> list = new ArrayList<>();
for (int i = 300; i < 400; i++){
    list.add(i);
}
```

But this comes with a performance cost. If you think about it, boxed values are essentially a wrapper around primitive types and are stored on the heap. Therefore boxed values use more memory and require additional memory lookups to fetch the wrapped primitive value.

Java 8 brings a specialized version of the functional interfaces we described earlier in order to avoid autoboxing operations when the inputs or outputs are primitives. For example, in the following code, using an `IntPredicate` avoids a boxing operation of the value 1000, whereas using a `Predicate<Integer>` would box the argument 1000 to an `Integer` object.

```
public interface IntPredicate{
    public boolean test(int t);
}

IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000); // true (no boxing)

Predicate<Integer> oddNumbers = (Integer i) -> i % 2 == 1;
oddNumbers.test(1000); // false (boxing)
```

In general, functional interfaces that have a specialization for the input type parameter have their names preceded by the appropriate primitive type, for example, `DoublePredicate`, `IntConsumer`, `LongBinaryOperator`, `IntFunction`, and so on. The `Function` interface has also variants for the output type parameter: `ToIntFunction<T>`, `IntToDoubleFunction`, and so on.

Table 3.2 gives a summary of the most common functional interfaces available in the Java API and their function descriptors. Keep it mind that they are only a starter kit. You can always make your own if needed! The notation `(T,U) -> R` shows how to think about a function descriptor. The left side is a list representing the types of the arguments. In this case it represents a function with two arguments of respectively generic type `T` and `U` and that has a return type of `R`.

Table 3.2 Common functional interfaces in Java 8

Functional interface	Function descriptor	Primitive specializations
`Predicate<T>`	`T -> Boolean`	`IntPredicate, LongPredicate, DoublePredicate`
`Consumer<T>`	`T -> void`	`IntConsumer, LongConsumer, DoubleConsumer`
`Function<T, R>`	`T -> R`	`IntFunction,`

		IntToDoubleFunction, IntToLongFunction, LongFunction, LongToDoubleFunction, LongToIntFunction, ToIntFunction, ToDoubleFunction, ToLongFunction
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<L, R>	(L, R) -> boolean	
BiConsumer<T, U>	(T, U) -> void	
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction, ToLongBiFunction, ToDoubleBiFunction

You've now seen a lot of functional interfaces that can be used to describe the signature of various lambda expressions. To check your understanding so far, have a go at Quiz 3.4.

Quiz 3.4: Functional interfaces

What functional interfaces would you use for the following function descriptors (that is, signature of a lambda expression)? As a further exercise, come up with valid lambda expressions that you can use with these functional interfaces.

1. `T -> R`
2. `(int, int) -> int`
3. `T -> void`
4. `() -> T`
5. `(T, U) -> R`

Answers:

1.

`Function<T, R>` is a good candidate. It's typically used for converting an object of type `T` into an object of type `R` (for example, `Function<Apple, Integer>` to extract the weight of an apple).

2.
`IntBinaryOperator` has one single abstract method called `applyAsInt` representing a function descriptor `(int, int) -> int`.
3.
`Consumer<T>` has one single abstract method called `accept` representing a function descriptor `T -> void`.
4.
`Supplier<T>` has a single abstract method called `get` representing a function descriptor `() ->T`. Alternatively, `Callable<T>` has also a single abstract method called `call` representing a function descriptor `() -> T`.
5.
`BiFunction<T, U, R>` has a single abstract method called `apply` representing a function descriptor `(T, U) -> R`.

To summaries the discussion about functional interfaces and lambdas, table 3.3 provides a summary of use cases, examples of lambdas, and functional interfaces that can be used.

Table 3.3 Examples of lambdas with functional interfaces

Use case	Example of lambda	Context
A Boolean expression	`(List<String> list) -> list.isEmpty()`	`Predicate<List<String>>`
Creating objects	`() -> new Apple(10)`	`Supplier<Apple>`
Consuming from an object	`(Apple a) -> System.out.println(a.getWeight())`	`Consumer<Apple>`
Select/extract from an object	`(String s) -> s.length()`	`Function<String, Integer>` or `ToIntFunction<String>`
Combine two values	`(int a, int b) -> a * b`	`BinaryOperator<Integer>`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`	`Comparator<Apple>` or `BiFunction<Apple, Apple, Integer>` or `ToIntBiFunction<Apple, Apple>`

What about exceptions, lambdas and functional interfaces?

Note that none of the functional interfaces allows for a checked exception to be thrown. You have two options if you need a lambda expression to throw an exception: define your own functional interface that declares the checked exception or wrap the lambda with a try/catch block.

For example, in section 3.3 we introduce a new functional interface `BufferedReaderProcessor` that explicitly declared an `IOException`:

```
@FunctionalInterface
public interface BufferedReaderProcessor {
  public String process(BufferedReader b) throws IOException;
}

BufferedReaderProcessor p = (BufferedReader br) -> br.readLine();
```

However, you may be using an API that expects a functional interface such as `Function<T, R>` and there's no option to create your own (you will see in the next chapter that the Streams API makes heavy use of the funcitonal interfaces from table 3.2). In this case you can explicitly catch the checked exception:

```
Function<BufferedReader, String> f =
  (BufferedReader b) -> {
    try {
      return b.readLine();
    }
    catch(IOException e) {
      throw new RuntimeException(e);
    }
  };
```

You've now seen how to create lambdas and where and how to use them. Next, we'll explain how lambdas are type checked by the compiler and rules you should be aware of, such as lambdas referencing local variables inside their body and void-compatible lambdas.

3.5 Type checking, type inference, and restrictions

When we first mentioned lambda expressions, we said that they let you generate an instance of a functional interface. Nonetheless, a lambda expression itself doesn't contain the information about which functional interface it's implementing. So you may be wondering what the actual type of a lambda is.

3.5.1 Type checking

The type of a lambda is deduced from the context in which the lambda is used. The type expected for the lambda expression inside the context (e.g. a method parameter it's passed to or a local variable that it's assigned to) is called *the target type*. Let's look at an example to see what happens behind the scene when you use a lambda expressions. Using lambda expression we can re-factor the following code as follows:

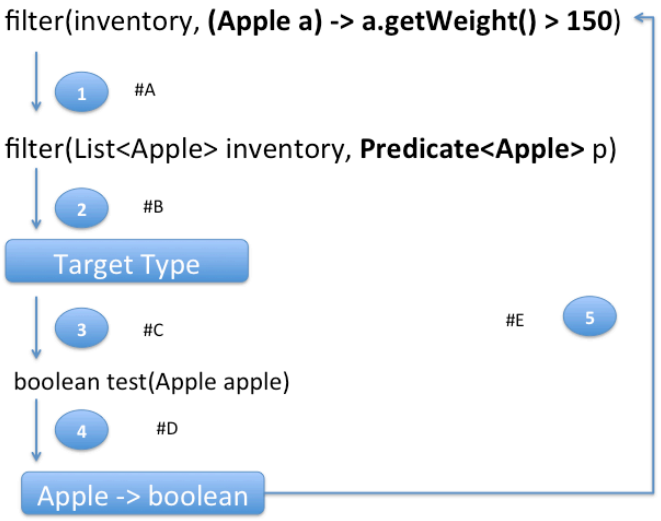
BEFORE:

```
List<Apple> heavierThan150g = filter(inventory, new ApplePredicate() {
    public boolean test(Apple a){
        return a.getWeight() > 150;
    }
} );
```

AFTER:

```
List<Apple> heavierThan150g =
        filter(inventory, (Apple a) -> a.getWeight() > 150);
```

Figure 3.4 summarizes the type checking process.



#A: What's the context in which the lambda is used? Let's first look up the definition of `filter`.
#B: Cool, the target type is `Predicate<Apple> (T is bound to Apple)`!
#C: What's the abstract method in the `Predicate<Apple>` interface?
#D: Cool, it's `test`, which takes an `Apple` and returns a `boolean`!
#E: The function descriptor `Apple -> boolean` matches the signature of the lambda! It takes an `Apple` and returns a `boolean`, so our code type checks.

Figure 3.4 Deconstructing the type-checking process of a lambda expression

- First, you look up the declaration of the `filter` method.
- Second, it expects as second formal parameter an object of type `Predicate<Apple>` (the target type).
- Third, `Predicate<Apple>` is a functional interface defining one single abstract method called `test`.
- Fourth, the method `test` describes a function descriptor that accepts an `Apple` and returns a `boolean`.
- Finally, any actual argument to the `filter` method needs to match this requirement.

The code is valid because the lambda expression that we're passing also takes an `Apple` as parameter and returns a `boolean`!

3.5.2 Same lambda, different functional interfaces

Because of the idea of *target typing*, the same lambda expression can be associated with different functional interfaces, if they have a compatible abstract method signature.
For example, both interfaces `Callable` and `PrivilegedAction` described earlier represent a function, which accepts nothing and returns a generic type `T`. The following two assignments are therefore valid:

```
Callable<Integer> c = () -> 42;
PriviledgedAction<Integer> p = () -> 42;
```

In this case the first assignment has target type `Callable<Integer>` and the second assignment has target type `PrivilegedAction<Integer>`.
In table 3.3 we showed a similar example, the same lambda can be used with multiple different functional interfaces:

```
Comparator<Apple> c1 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
ToIntBiFunction<Apple, Apple> c2 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
BiFunction<Apple, Apple, Integer> c3 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Diamond operator

Readers familiar with Java evolution will recall that Java 7 had already introduced the idea of types being inferred from context with generics inference using the diamond operator (<>) (this idea can actually be seen even earlier with generic methods). A given class instance expression can appear in two or more different contexts and the appropriate type argument will be inferred as exemplified here:

```
List<String> listOfStrings = new ArrayList<>();
List<Integer> listOfIntegers = new ArrayList<>();
```

Special void-compatibility rule

If a lambda has a statement expression as body, it's compatible with a function descriptor that returns void (provided the parameter list is compatible too, of course). For example, both of the following lines are legal even though the method add of a `List` returns a `boolean` and not `void` as expected in the `Consumer` context (`T -> void`):

```
// Predicate has a boolean return
Predicate<String> p = s -> list.add(s);
// Consumer has a void return
Consumer<String> b = s -> list.add(s);
```

By now you should have a good understanding of when and where you're allowed to use lambda expressions. They can get their target type from an assignment context, method invocation context (parameters and return), and additionally a cast context. To check your knowledge, try Quiz 3.5.

Quiz 3.5: Type checking – Why won't the following code compile?

How could you fix the problem?

```
Object o = () -> {System.out.println("Tricky example"); };
```

The context of the lambda expression is `Object` (the target type). But `Object` isn't a functional interface. To fix this you can provide an explicit target type by casting the lambda expression:

```
Object o = (Runnable) () -> {System.out.println("Tricky example"); };
```

You've seen how the target type can be used to check whether a lambda can be used in a particular context. It can also be used to do something slightly different: infer the types of the parameters of a lambda!

3.5.3 Type inference

You can simplify your code one step further. Because the Java compiler deduces what functional interface to associate with a lambda expression from its surrounding context (the target type), it can also deduce an appropriate signature for the lambda, because the function descriptor is available through the target type. The benefit is that the compiler has access to the types of the parameters of a lambda expression, and they can be omitted in the lambda

syntax. In other words, the Java compiler infers the types of the parameters of a lambda as shown here:8

```
List<Apple> greenApples =
        filter(inventory, a -> "green".equals(a.getColor())); #A
```

#A: No explicit type on the parameter a

The benefits of code readability are more noticeable with lambda expressions that have several parameters. For example, here's how to create a `Comparator` object:

```
Comparator<Apple> c =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()); # A

Comparator<Apple> c =
  (a1, a2) -> a1.getWeight().compareTo(a2.getWeight()); # B
```

 # A: Without type inference
 # B: With type inference

Note that sometimes its more readable to include the types explicitly, and sometimes more readable to exclude them. There is no one rule for which is better; developers must make their own choices about what makes their code most readable.

3.5.4 Capturing variables within a lambda

All the lambda expressions we've shown so far only used their arguments inside their body. But lambda expressions are also allowed to use *free variables* (variables that are not the parameters and defined in an outer scope) just like anonymous classes can. They're called *capturing lambdas*. For example, the following lambda captures the variable `portNumber`:

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
```

Nonetheless, there's a small twist: there are some restrictions on what you can do with these variables. Lambdas are allowed to capture instance variables and static variables without restrictions. But captured local variables have to be explicitly declared final or are effectively final. In other words, lambda expressions can capture local variables that are assigned to only once. (Note: capturing an instance variable can be seen as capturing the final local variable `this`). For example, the following code doesn't compile because the variable `portNumber` is assigned to twice:

```
int portNumber = 1337;
// error: local variables referenced from a lambda expression must be final or effectively final
Runnable r = () -> System.out.println(portNumber);
portNumber = 31337;
```

8 Note also that when a lambda has just one parameter whose type is inferred, the parentheses surrounding the parameter name can also be omitted.

RESTRICTIONS ON LOCAL VARIABLES

So you may be asking yourself, why do local variables have these restrictions?

First, there's a key difference in how instance and local variables are implemented. Instance variables are stored on the heap, whereas local variables live on the stack. If a lambda could access the local variable directly and the lambda were used in a thread, then the thread using the lambda could try to access the variable after the thread which allocated the variable had de-allocated it! Hence Java implements accesses to a free local variable as accesses to a copy of it, rather than accesses to the original variable. This makes no difference if the local variable is only assigned to once – hence the above restriction.

Second, this restriction also discourages typical imperative programming patterns that mutate an outer variable, which as we explain in later chapter prevent easy parallelization. (The restriction still allows a lambda to mutate instance variables, which cannot be de-allocated by another thread, but note that such mutation runs the risk of needing locking if the lambda is used in another thread, such as in the Stream API discussed in Chapter 4.)

The more hacker type of reader might note that there's always a workaround. Because arrays live on the heap and a variable stores a reference to it, you could create a similar effect using a local variable declared final that references an array on the heap. You could then mutate the elements inside that array. Hence the following code is legal but a bad idea: it is thread-unsafe and hopefully ugly enough to discourage its use!

```
final int[] wrappedNumber = new int[1];
Runnable r = () -> System.out.println(wrappedNumber[0]);
wrappedNumber[0] = 31337; // OK
```

Note that the Java compiler could use a similar trick to generate code like this behind the scenes, but this would result in a big mess—every local variable would have a slot on the heap, which would impact application performance.

Closure

You may have heard of the term *closure* and may be wondering whether lambdas meet the definition of a closure (not to be confused with the Clojure programming language). To put it scientifically, a *closure* is essentially an instance of a function that can reference nonlocal variables of that function with no restrictions. For example, a closure could be passed as argument to another function. It could also *access and modify* variables defined outside its scope. Now, Java 8 lambdas and anonymous classes do something similar to closures: they can be passed as argument to methods and can access variables outside their scope. But they have a restriction: they can't modify the content of local variables of a method in which the lambda is defined. Those variables have to be implicitly final. It helps to think that lambdas close over *values* rather than *variables*. As explained previously, this restriction exists because local variables live on the stack and are implicitly confined to the thread they're in. Allowing capture of mutable local variables opens new thread-unsafe possibilities,

which are undesirable (instance variables are fine because they live on the heap, which is shared across threads).

We now describe another feature that you'll see in Java 8 code: *method references*. They can be seen as shorthand versions of certain lambdas.

3.6 Method references

Method references let you reuse existing method definitions and pass them just like lambdas. In some cases they can look more readable and feel more natural than using lambda expressions. Here's our sorting example written with a method reference and a bit of help from the updated Java 8 API (we explore this example in more detail in section 3.7):

BEFORE:

```
inventory.sort((Apple a1, Apple a2)
                   -> a1.getWeight().compareTo(a2.getWeight()));
```

AFTER (USING A METHOD REFERENCE AND `java.util.Comparators.comparing`):

```
inventory.sort(comparing(Apple::getWeight)); #A
#A: Your first method reference!
```

3.6.1 In a nutshell

Why should you care about method references? Method references can be seen as shorthand to lambdas only calling a specific method. The basic idea is that if a lambda represents "call this method directly", it's best to refer to the method by name, rather than by a description of how to call it. Indeed, a method reference lets you create a lambda expression from an existing method implementation. But by referring to a method name explicitly, your code *can gain better readability*. How does it work? When you need a method reference, the target reference is placed before the delimiter `::` and the name of the method is provided after it. For example, `Apple::getWeight` is a method reference to `getWeight()` defined in the `Apple` class. It can be seen as a shorthand for the lambda expression `(Apple a) -> a.getWeight()`. Table 3.4 below gives a couple more examples of simple method references.

Table 3.4 Examples of lambdas and method references equivalent

Lambda	Method reference equivalent
`(Apple a) -> a.getWeight()`	`Apple::getWeight`
`() -> Thread.currentThread().dumpStack()`	`Thread.currentThread()::dumpStack`
`(str, i) -> str.substring(i)`	`String::substring`
`(String s) -> System.out.println(s)`	`System.out::println`

Method references can be seen as a syntactic sugar for lambdas that refer only to a single method because you have to write less to express the same thing.

RECIPE FOR CONSTRUCTING METHOD REFERENCES

There are three main kinds of method references.

1. A method reference to a *static method* (for example, the method `parseInt` of `Integer`, written `Integer::parseInt`)
2. A method reference to an *instance method of an arbitrary type* (for example, the method `length` of a `String`, written `String::length`)
3. A method reference to an *instance method of a specific object* (for example, suppose you have an object `expensiveTransaction` from class `Transaction` with an instance method `getValue()`, you can write `expensiveTransaction::getValue`)

Note that there are also special forms for constructors, array constructors and super-calls. The shorthand rules follow a simple recipe, shown in figure 3.5.

Figure 3.5 Recipes for constructing method references for three different types of lambda expressions

Let's apply method references in a concrete example. Say you'd like to sort a `List` of strings, ignoring case differences. The `sort` method on a `List` expects a `Comparator` as parameter.

You saw earlier that `Comparator` describes a function descriptor with the signature `(T,T) -> int`. You can define a lambda expression that leverages the method `compareToIgnoreCase` in the `String` class as follows (note that `compareToIgnoreCase` is already predefined in the `String` class):

```
List<String> str = Arrays.asList("a","b","A","B");
str.sort((s1,s2) -> s1.compareToIgnoreCase(s2));
```

The lambda expression has a signature compatible with the function descriptor of `Comparator`. Using the recipes described previously, the example can also be written using a method reference as follows:

```
List<String> str = Arrays.asList("a","b","A","B");
str.sort(String::compareToIgnoreCase);
```

Note that the compiler goes through a similar type checking process as for lambda expressions to figure out whether a method reference is valid with a given functional interface: the signature of the method reference has to match the type of the context.

To check your understanding of method references, have a go at Quiz 3.6!

Quiz 3.6: Method references

What are equivalent method references for the following lambda expressions?

1.

```
Function<String, Integer> stringToInteger =
                (String s) -> Integer.parseInt(s);
```

2.

```
BiPredicate<List<String>, String> contains =
                (list, element) -> list.contains(element);
```

Answers:

1. This lambda expression forwards its argument to the static method `parseInt` of `Integer`. This method takes a `String` to parse and returns an `Integer`. As a result, the lambda can be rewritten using recipe #1 as follows:

```
Function<String, Integer> stringToInteger = Integer::parseInt;
```

2. This lambda uses its first argument to call the method `contains` on it. Because the first argument is of type `List`, you can use recipe #2 as follows:

```
BiPredicate<List<String>, String> contains = List::contains;
```

This is because the target type describes a function descriptor `(List<String>, String) -> boolean` and `List::contains` can be unpacked to that function descriptor.

We only showed so far how to reuse existing method implementations and create method references. But you can do something similar with constructors of a class.

3.6.2 Constructor references

You can create a reference to an existing constructor using its name and the keyword new as follows: `ClassName::new`. It works similarly to a reference to a static method. For example, Suppose there is a zero-argument constructor. This fits the signature `() -> Apple` of `Supplier`, we can do this:

```
Supplier<Apple> c1 = Apple::new;
Apple a1 = c1.get();
```

If we have a constructor with signature `Apple(int weight)` it fits the signature of the `Function` interface, so we can do this:

```
Function<Integer, Apple> c2 = Apple::new;
Apple a2 = c2.apply(110);
```

In the following code, each element of a `List` of integers is passed to the constructor of `Apple` using a similar `map` method we defined earlier, resulting in a `List` of apples with different weights:

```
List<Integer> weights = Arrays.asList(7, 3, 4, 10);
List<Apple> apples = map(weights, Apple::new);

public static List<Apple> map(List<Integer> list,
                              Function<Integer, Apple> f){
    List<Apple> result = new ArrayList<>();
    for(Integer e: list){
        result.add(f.apply(e));
    }
    return result;
}
```

If we have a two-argument constructor `Apple(String color, int weight)` it fits the signature of the `BiFunction` interface, so we can do this:

```
BiFunction<String, Integer, Apple> c2 = Apple::new;
Apple c3 = c3.apply("green", 110);
```

The capability of referring to a constructor without instantiating it enables interesting applications. For example, you can use a `Map` to associate constructors with a string value. You can then create a method `giveMeFruit` that given a `String` and an `Integer` can create different types of fruits with different weights:

```
Map<String, Function<Integer, Fruit>> map = new HashMap<>();
static {
    map.put("apple", Apple::new);
    map.put("orange", Orange::new);
```

```
    // etc...
}

public static Fruit giveMeFruit(String fruit, Integer weight){
    return map.get(fruit.toLowerCase()).apply(weight);
}
```

To check your understanding of method and constructor references, try out Quiz 3.7.

Quiz 3.7: Constructor references

You saw how to transform zero, one and two argument constructors into a constructor reference? What would you need to do in order to use a constructor reference for a three-argument constructor such as `Color(int, int, int)`?

Answers:

You saw that the syntax for a constructor reference is `ClassName::new`, so in this case it is `Color::new`. However, we need a functional interface that will match the signature of that constructor reference. Since there isn't one in the functional interface starter set, you can create your own:

```
public interface TriFunction<T, U, V, R>{
    R apply(T t, U u, V v);
}
```

And you can now use the constructor reference as follows:

```
TriFunction<Integer, Integer, Integer, Color> = Color::new;
```

We've gone through a lot of new information: lambdas, functional interfaces, and method references. Let's put it all in practice in the next section!

3.7 Putting it all into practice!

To wrap up this chapter and all that we've discussed on lambdas, we start again with our initial problem of sorting a list of `Apples` with different ordering strategies and show how you can progressively evolve a naïve solution into a concise solution, using all the concepts and features explained in this chapter: *behavior parameterization*, *anonymous classes*, *lambda expressions*, and *method references*. The final solution we'll work toward is this (note that all source code is available on the book's web page):

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

3.7.1 Step 1: Passing code

You're lucky: the Java API already provides you with a `sort` method available on `List` so you don't have to implement it. So the hard part is done! But how can you pass an ordering strategy to the `sort` method? Well, the `sort` method has the following signature:

```
void sort(Comparator<? super E> c)
```

It expects a `Comparator` object as argument to `Apples`! This is how you can pass different strategies in Java: they have to be wrapped in an object. We say that the *behavior* of sort is *parameterized*: its behavior will be different based on different ordering strategies passed to it.

So your first solution looks like this:

```
public class AppleComparator implements Comparator<Apple> {
        public int compare(Apple a1, Apple a2){
                return a1.getWeight().compareTo(a2.getWeight());
        }
}

inventory.sort(new AppleComparator());
```

3.7.2 Step 2: Anonymous class

Rather than implementing `Comparator` for the purpose of instantiating it once, you saw that you could use an *anonymous class* to improve your solution:

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

3.7.3 Step 3: Lambda expressions

But our current solution is still verbose. Java 8 introduces lambda expressions, which provides a lightweight syntax to achieve the same goal: *passing code*. You saw that lambda expressions can be used where a *functional interface* is expected. As a reminder, a functional interface is an interface defining only one abstract method. The signature of the abstract method (called *function descriptor*) can describe the signature of a lambda expression. In this case, the `Comparator` represents a function descriptor `(T, T) -> int`. Because we're using apples, it represents more specifically `(Apple, Apple) -> int`.

Your new improved solution looks therefore as follows:

```
inventory.sort((Apple a1, Apple a2)
                -> a1.getWeight().compareTo(a2.getWeight())
);
```

We explained that the Java compiler could *infer the types* of the parameters of a lambda expression by using the context in which the lambda appears. So you can rewrite your solution as follows:

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

Can you make your code even more readable? There exists a static helper method in `Comparator` called `comparing`, which takes a `Function` extracting a `Comparable` key and produces a `Comparator` object (we explain why interfaces can have static methods in chapter 6). It can be used as follows (note that we now pass a lambda with only one argument: the lambda specifies how to extract the key to compare with from an apple):

```
Comparator<Apple> c = Comparator.comparing((Apple a) -> a.getWeight());
```

You can now rewrite your solution in a slightly more compact form:

```
import static java.util.Comparator.comparing;
inventory.sort(comparing((a) -> a.getWeight()));
```

3.7.4 Step 4: Method references

We explained that method references are syntactic sugar for lambda expressions that forwards their arguments. You can use a method reference to make your code slightly less verbose (assuming a static import of `java.util.Comparator.comparing`):

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

Congratulations, this is your final solution! Why is this better than code prior to Java 8? It's not just because it is shorter; it is also obvious what it means, and the code reads like the problem statement: "sort inventory comparing the weight of the apples".

3.8 Useful methods to compose lambda expressions

Several functional interfaces in the Java 8 API contain convenient methods. Specifically, many functional interfaces such as `Comparator`, `Function` and `Predicate` that are used to pass lambda expressions provide methods that allow composition. What does this mean? In practice it means you can combine several simple lambda expressions to build more complicated ones. For example, you can combine two predicates into a larger predicate that perform an "or" operator between the two predicate. Moreover, can also compose functions such as the result of one becomes the input of another function. You may wonder how is it possible that there are additional methods in a functional interface (after all this goes against the definition of a functional interface!)? The trick is that the methods that we will introduce are called *default methods* (i.e. they are not abstract methods). We explain them in detail in chapter 8. For now, just trust us and read chapter 8 later when you want to find out more about what default methods are and what you can do with them.

3.8.1 Composing Comparators

You've seen that we can use the static method `Comparator.comparing` to return a `Comparator` based on a `Function` that extracts a key for comparison.

REVERSED ORDER

What if we wanted to sort the apples by decreasing weight? There's no need to create a different instance of a `Comparator`. The interface includes a default method `reverse` that imposes the reverse ordering of a given comparator. So we can simply modify our previous example to sort the apples by decreasing weight by re-using our initial `Comparator`:

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

CHAINING COMPARATORS

This is all nice but what if we find two apples that have the same weight. Which apple should have priority in the sorted list? You may want to provide a second `Comparator` to further refine the comparison. For example, after two apples are compared based on the weight, you may want to sort them by country of origin. The `thenComparing` method allows you to do just that. It takes a function as parameter (just like the method `comparing`) and provides a second `Comparator` if two objects are considered equal using the initial `Comparator`. We can solve our problem elegantly again:

```
inventory.sort(comparing(Apple::getWeight)
               .reverse()
               .thenComparing(Apple::getCountry));
```

3.8.2 Composing Predicates

The `Predicate` interface includes three methods that let you re-use an existing `Predicate` to create more complicate ones: `negate`, `and`, `or`.
For example, you can use the method `negate` to return the negation of a `Predicate`. For example, an apple that is not red:

```
Predicate<Apple> notRedApple = redApple.negate();
```

You may want to combine two lambdas to say that an apple is both red and heavy with the `and` method:

```
Predicate<Apple> redAndHeavyApple =
    redApple.and(a -> a.getWeight() > 150);
```

We combine the resulting predicate one step further to express apples that are red and heavy (above 150g) or just green apples.

```
Predicate<Apple> redAndHeavyAppleOrGreen =
    redApple.and(a -> a.getWeight() > 150)
            .or(a -> "green".equals(a.getColor()));
```

Why is this great? From simpler lambda expressions we are able to represent more complicated lambda expressions that still read like the problem statement!

3.8.3 Composing Functions

Finally, you can also compose lambda expressions represented by the `Function` interface. The `Function` interfaces comes with two default methods for this: `andThen` and `compose` that both return an instance of `Function`.

The method `andThen` returns a function that applies a given function first to an input, and then applies another function to the result of that application. For example, given a function `f` that increments a number (`x -> x + 1`) and another function `g` that multiples a number by two, we can combine them to create a function `h` that first increment a number and then multiplies the result by two:

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.andThen(g); #A
int result = h.apply(1); #B
```

#A: in maths we would write g(f(x)) or (g o f)(x)
#B: this returns 4

You can also use the method `compose` similarly to say to first apply the function given as argument to `compose` and then apply the function to the result. For example, in the example above using `compose` would mean `f(g(x))` instead of `g(f(x))` using `andThen`:

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g); #A
int result = h.apply(1); #B
```

#A: in maths we would write f(g(x)) or (f o g)(x)
#B: this returns 3

Figure 3.6 below illustrate the difference between `andThen` and `compose`.

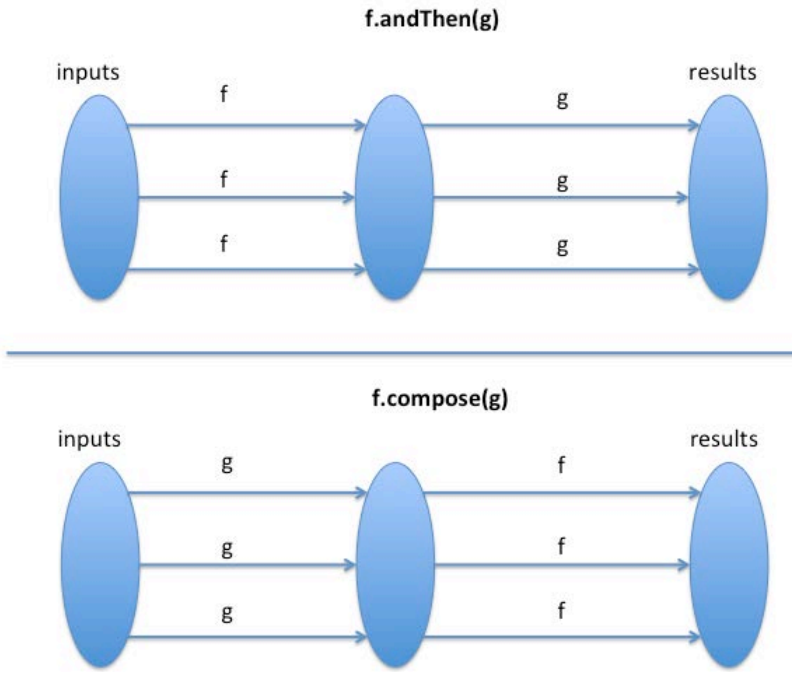


Figure 3.6 Using andThen vs. compose

This all sounds a little bit too abstract. How can you use these in practice? Let's say you have various utility methods that do text transformation on a letter represented as a `String`:

```
public class Letter{
    public static String addHeader(String text){
        return "From Raoul, Mario and Alan: " + text;
    }

    public static String addFooter(String text){
        return text + " Kind regards";
    }

    public static String checkSpelling(String text){
        return text.replaceAll("labda", "lambda");
    }
}
```

You can now create various transformation pipelines by composing the utility methods. For example, creating a pipeline that first adds a header, then checks spelling and finally adds a footer as illustrated in figure 3.7:

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
  = addHeader.andThen(Letter::checkSpelling)
             .andThen(Letter::addFooter);
```

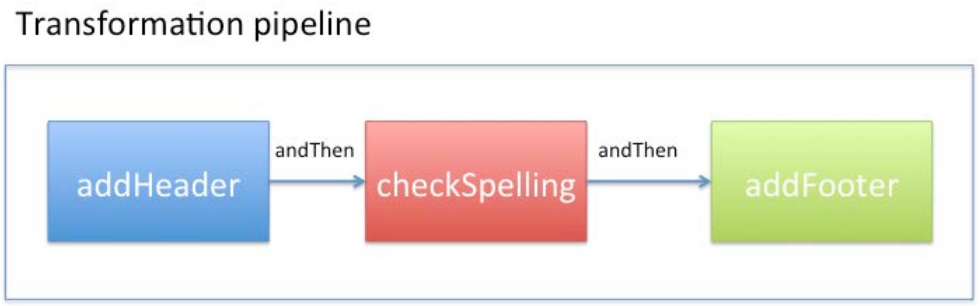


Figure 3.7 A transformation pipeline using andThen

A second pipeline might be only adding a header and footer without checking for spelling:

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
  = addHeader.andThen(Letter::addFooter);
```

3.9 Similar ideas from mathematics

If you feel comfortable with school mathematics, then this section gives another viewpoint of the idea of lambda expressions and passing around functions. If you're not comfortable with school mathematics, then just skip it; nothing else in the book depends on it.

3.9.1 Integration

Suppose you have a (mathematical, not Java) function `f`, perhaps defined by

$$f(x) = x + 10$$

Then, one question that's often asked (at school, in engineering degrees) is that of finding the area beneath the function when drawn on paper (counting the x-axis as the zero line). For example, you write

$$\int_3^7 f(x)dx \qquad \text{or} \qquad \int_3^7 (x+10)dx$$

for the area shown in figure 3.8.

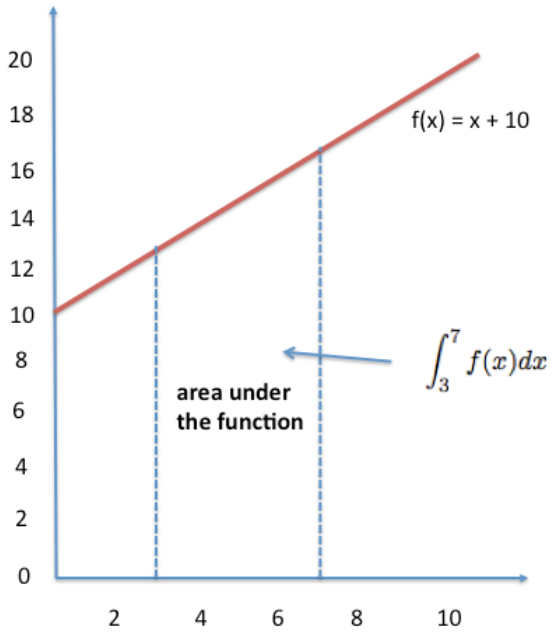


Figure 3.8 Area under the function `f(x) = x + 10` for `x` from 3 to 7

In this example, the function `f` is a straight line, and so you can easily work out this area by the trapezium method (essentially drawing triangles) to discover is the solution:

$$\frac{1}{2} \times ((3+10)+(7+10)) \times (7-3) \quad = \quad 60$$

Now, how might you express this in Java? Your first problem is reconciling the strange notation like the integration symbol or dy/dx with familiar programming language notation.

Indeed, thinking from first principles you need a method, perhaps called `integrate`, which takes three arguments: one is `f`, and the others are the limits (3.0 and 7.0 here).

Thus, you want to write in Java something that looks like this, where the *function* `f` is just passed around:

```
integrate(f, 3, 7)
```

Note that you can't write something as simple as

```
integrate(x+10, 3, 7)
```

for two reasons. First, the scope of x is unclear, and the second is that this would pass a value of x+10 to integrate instead of passing the function f.

Indeed, the secret role of dx in the mathematics is to say "that function taking argument x whose result is x+10."

3.9.2 Connecting to Java 8 lambdas

Now, as we mentioned earlier, Java 8 uses the notation (double x) -> x+10 (a lambda expression) for exactly this purpose; hence you can write

```
integrate((double x) -> x + 10, 3, 7)
```

or

```
integrate((double x) -> f(x) 3, 7)
```

or, using a method reference as mentioned earlier, simply

```
integrate(C::f, 3, 7)
```

if C is a class containing f as a static method.

The idea is that you're passing the code for f to the method integrate.

You many now wonder how you would write the method integrate itself. We'll continue to suppose either that f is a linear function (straight line). You'd probably like to write in a form similar to mathematics:

```
public double integrate((double->double)f, double a, double b) { #A
      return (f(a)+f(b))*(b-a)/2.0
}
```

#A: incorrect Java code! (you can't write functions as you do in mathematics)

But because lambda expressions can only be used in a context expecting a functional interface (in this case Function), you'll have to write it this way:

```
public double integrate(Function<Double, Double> f, double a, double b) {
      return (f.apply(a) + f.apply(b)) * (b-a) / 2.0;
 }
```

As a side remark, it's a bit of a shame you have to write f.apply(a) instead of just f(a) as in mathematics, but Java just can't get away from the view that everything is an object—instead of the idea of a function being truly independent!

3.10 Summary

Following are the key concepts you should take away from this chapter:

- A *lambda expression* can be understood as a kind of anonymous function: it doesn't have a name, but it has a list of parameters, a body a return type and also possibly a list of exceptions that can be thrown.

- Lambda expressions let you pass code concisely.
- A *functional interface* is an interface that declares exactly one abstract method.
- Lambda expressions can be used only where a functional interface is expected.
- Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and *treat the whole expression as an instance of a functional interface.*
- Java 8 comes with a list of common functional interfaces in the `java.util.function` package, which includes `Predicate<T>`, `Function<T, R>`, `Supplier<T>`, `Consumer<T>`, and `BinaryOperator<T>`.
- There are primitive specializations of common generic functional interfaces such as `Predicate<T>` and `Function<T, R>` that can be used to avoid boxing operations: `IntPredicate`, `IntToLongFunction`, and so on.
- The execute around pattern (that is, you need to execute a bit of behavior in the middle of code that's always required in a method, for example, resource allocating and cleaning up) can be used with lambdas to gain additional flexibility and reusability.
- The type expected for the lambda expression inside a context is called the target type.
- Method references let you reuse an existing method implementation and pass it around directly.
- Functional interfaces such as `Comparator`, `Predicate` and `Function` have several default methods that can be used to combine lambda expressions.