3

Lambda expressions

This chapter covers

- Lambdas in a nutshell
- Where and how to use lambdas
- 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 define a block of code that represents a behavior and then pass it around. You can decide to run that block of code when a certain event happens (for example, a click on a button) or at certain points in an algorithm (for example, a predicate such as "only apples heavier than 150 g" in the filtering algorithm or the customized comparison operation in sorting). In general, using this concept you can write code that's more flexible and reusable.

But you saw that using anonymous classes to represent different behaviors is unsatisfying: it's 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 behavior 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 as arguments to a method as you can with 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 bring together all the concepts taught into a concrete example: we take the sorting example shown in chapter 2 and gradually improve it using lambda expressions and method references to make it more concise and readable. This chapter is important in itself and also because you'll use lambdas extensively throughout the book.

3.1 Lambdas in a nutshell

A *lambda expression* can be understood as a concise representation of an 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. 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 exceptions 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.

If you're wondering where the term *lambda* comes from, it originates from a system developed in academia called *lambda calculus*, which is used to describe computations.

Why should you care about lambda expressions? 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. Lambdas technically don't let you do anything that you couldn't do prior to Java 8. But you no longer have to write clumsy code using anonymous classes to benefit from behavior parameterization! Lambda expressions will encourage you to adopt the style of behavior parameterization that we described in the previous chapter. The net result is that your code will be clearer and more flexible. For example, using a lambda expression you 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 explain all the pieces soon. For now, note that you're literally passing only the code that's really needed to compare two apples using their weight. It looks like you'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:

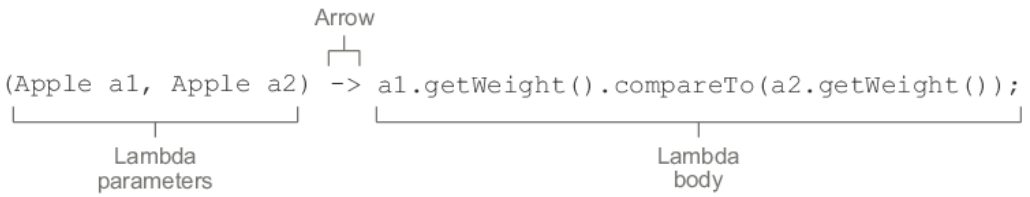

Figure 3.1 A lambda expression is composed of parameters, an arrow, and a body.

- *A list of parameters*—In this case it mirrors the parameters of the `compare` method of a `Comparator`—two `Apple`s.
- *An arrow*—The arrow `->` separates the list of parameters from the body of the lambda.
- *The body of the lambda*—Compare two `Apple`s using their weights. The expression is considered the lambda's return value.

To illustrate further, the following listing shows five examples of valid lambda expressions in Java 8.

Listing 3.1 Valid lambda expressions in Java 8

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

(1) **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.**
(2) **The second lambda expression has one parameter of type Apple and returns a boolean (whether the apple is heavier than 150 g).**
(3) **The third lambda expression has two parameters of type int with no return (void return). Note that lambda expressions can contain multiple statements, in this case two.**

4. The fourth lambda expression has no parameter and returns an int.
5. 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. JavaScript has a similar syntax. The basic syntax of a lambda is either (referred to as an *expression-style* lambda)

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

or (note the curly braces for statements, this lambda is often called a *block-style* lambda)

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

As you can see, lambda expressions follow a simple syntax. Working through Quiz 3.1 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?

1) `() -> {}`
2) `() -> "Raoul"`
3) `() -> { return "Mario"; }`
4) `(Integer i) -> return "Alan" + i;`
5) `(String s) -> { "Iron Man"; }`

Answer:

4 and 5 are invalid lambdas; the rest are valid. Details:

1) This lambda has no parameters and returns `void`. It's similar to a method with an empty body: `public void run() { }`. Fun fact: this is usually called the burger lambda. Take a look at it from the side and you will see it has a burger shape with two buns.
2) This lambda has no parameters and returns a `String` as an expression.
3) This lambda has no parameters and returns a `String` (using an explicit return statement, within a block).
4) `return` is a control-flow statement. To make this lambda valid, curly braces 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 braces 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 example lambdas with examples of use cases.

Table 3.1 Examples of lambdas

Use case	Examples of lambdas
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, you assigned a lambda to a variable of type `Comparator<Apple>`. You could also use another lambda with the `filter` method you implemented in the previous chapter:

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

So where exactly can you use lambdas? You can use a lambda expression in the context of a functional interface. In the code shown here, you can pass a lambda as second argument to the method `filter` because it expects an object of type `Predicate<T>`, which is a functional interface. Don't worry if this sounds abstract; we now explain in detail what this means and what a functional interface is.

3.2.1 Functional interface

Remember the interface `Predicate<T>` you created in chapter 2 so you could parameterize the behavior of the `filter` method? It's a functional interface! Why? Because `Predicate` specifies only one abstract method:

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

In a nutshell, a *functional interface* is an interface that specifies exactly one abstract method. You already know several other functional interfaces in the Java API such as `Comparator` and `Runnable`, which we explored in chapter 2:

```
public interface Comparator<T> {                          ❶
    int compare(T o1, T o2);
}
```

```
public interface Runnable {                                    ❷
    void run();
}
public interface ActionListener extends EventListener {        ❸
    void actionPerformed(ActionEvent e);
}
public interface Callable<V> {                                 ❹
    V call() throws Exception;
}
public interface PrivilegedAction<T> {                         ❺
    T run();
}
```

❶ java.util.Comparator
❷ java.lang.Runnable
❸ java.awt.event.ActionListener
❹ java.util.concurrent.Callable
❺ java.security.PrivilegedAction

> **NOTE** You'll see in chapter 13 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 grasp the concept of a functional interface.

Quiz 3.2: Functional interface

Which of these interfaces are functional interfaces?

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

Answer:

Only `Adder` is a functional interface.

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

`Nothing` isn't a functional interface because it declares no abstract method at all.

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 (more technically speaking, an instance of a *concrete implementation* of the functional interface). You can achieve the same thing with an anonymous inner class, although it's 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");          (1)
Runnable r2 = new Runnable() {                                    (2)
    public void run() {
        System.out.println("Hello World 2");
    }
};
public static void process(Runnable r) {
    r.run();
}
process(r1);                                                      (3)
process(r2);                                                      (4)
process(() -> System.out.println("Hello World 3"));              (5)
```

1. **Using a lambda**
2. **Using an anonymous class**
3. **Prints "Hello World 1"**
4. **Prints "Hello World 2"**
5. **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`).[12]

We use a special notation throughout the chapter to describe the signatures of lambdas and functional interfaces. The notation `() -> void` represents a function with an empty list of parameters returning `void`. This is exactly what the `Runnable` interface represents. As another example, `(Apple, Apple) -> int` denotes a function taking two `Apple`s as parameters and returning an `int`. We will provide more information about function descriptors in section 3.4 and table 3.2 later in the chapter.

You may already be wondering how lambda expressions are type checked. We detail how the compiler checks whether a lambda is valid in a 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 expression has the

[12] **Some languages such as Scala provide explicit type annotations in their type system to describe the type of a function (called function types). Java reuses existing nominal types provided by functional interfaces and maps them into a form of function types behind the scenes.**

same signature as the abstract method of the functional interface. For instance, in our earlier example, 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.

Lambdas and void method invocation

While this may feel weird, the following lambda expression is valid:

```
process(() -> System.out.println("This is awesome"));
```

After all, System.out.println returns `void` so this is clearly not an expression! Why don't we have to enclose the body with curly braces like this?

```
process(() -> { System.out.println("This is awesome"); });
```

It turns out that there's a special rule for a void method invocation defined in the Java Language Specification. You do not have to enclose a single void method invocation in braces.

You may be wondering, "Why can we pass a lambda only where a functional interface is expected?" The language designers considered alternative approaches such as adding function types (a bit like the special notation we introduced to describe the signature of lambda expressions—we revisit this topic in chapters 20 and 21) to Java. But they chose this way because it fits naturally without increasing the complexity of the language. In addition, most Java programmers are already familiar with the idea of an interface with a single abstract method (for example, with event handling). The most important reason though is that functional interfaces were already extensively used before Java 8. This means that they provide a nice migration path for using lambda expressions. In fact, if you have been already using functional interfaces such as `Comparator` and `Runnable` or even your own interfaces that happen to define only a single abstract method, you can now leverage lambda expressions without changing your APIs. Try Quiz 3.3 to test your knowledge of where lambdas can be used.

Quiz 3.3: Where can you use lambdas?

Which of the following 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) The first example is valid because the lambda `() -> {}` has the 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) The second example is also 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) The third example is invalid because the lambda expression `(Apple a) -> a.getWeight()` has the signature `(Apple) -> Integer`, which is different from the signature of the method `test` defined in `Predicate<Apple>: (Apple) -> boolean`.

What about @FunctionalInterface?

If you explore the new Java API, you will notice that functional interfaces are *generally* annotated with `@FunctionalInterface` (we show an extensive list in section 3.4, where we explore how to use functional interfaces in depth). This annotation is used to indicate that the interface is intended to be a functional interface and is therefore useful for documentation. In addition, 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 method is available. Note that the `@FunctionalInterface` annotation isn't mandatory, but it's good practice to use it when an interface is designed for that purpose. You can think of it like the `@Override` notation to indicate that a method is overridden.

3.3 Putting lambdas into practice: the execute-around pattern

Let's look at an example of how lambdas, together with behavior parameterization, can be used in practice to make your code more flexible and concise. 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 surround 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 that you use Java 7's try-with-resources statement, which already simplifies the code, because you don't have to close the resource explicitly):

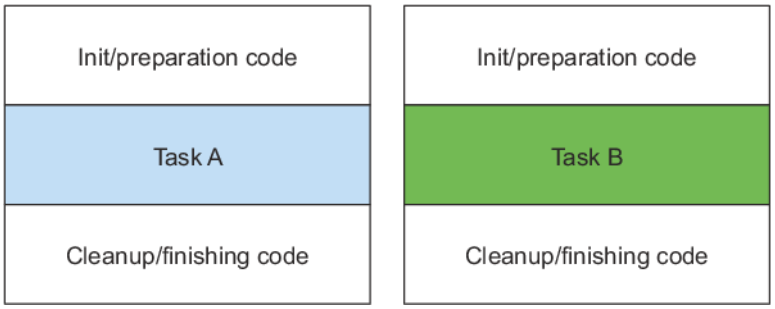

Figure 3.2 Tasks A and B are surrounded by the same redundant code responsible for preparation/cleanup.

```
public String processFile() throws IOException {
    try (BufferedReader br =
              new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();                                          (1)
    }
}
```

(1) This is the line that does useful work.

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 the word used most frequently? 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 `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 if you wanted to read two lines at once? 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: Use a functional interface to pass behaviors

We explained earlier that lambdas can be used only in the context of a functional interface. You need to create one that matches the signature `BufferedReader -> String` and that may throw an `IOException`. Let's call this interface `BufferedReaderProcessor`:

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

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

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

3.3.3 Step 3: Execute a behavior!

Any lambdas of the form `BufferedReader -> String` can be passed as arguments, because they match the signature of the `process` method defined in the `Buffered-ReaderProcessor` interface. You now need only a way to execute the code represented by the lambda inside the body of `processFile`. 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 String processFile(BufferedReaderProcessor p) throws IOException {
    try (BufferedReader br =
                  new BufferedReader(new FileReader("data.txt"))) {
        return p.process(br);                                          (1)
    }
}
```

(1) **Processing the BufferedReader object**

3.3.4 Step 4: Pass 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.

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

1

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

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

2

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

3

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

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

..
```

4

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

So far, we've shown 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 already available in the Java API such as `Comparable`, `Runnable`, and `Callable`, which you saw in section 3.2.

The Java library designers for Java 8 have helped you 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` that accepts an object of generic type `T` and returns a `boolean`. It's exactly the same one that you created 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, as shown in the following listing.

Listing 3.2 Working with a `Predicate`

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
public <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for(T t: list) {
        if(p.test(t)) {
            results.add(t);
        }
    }
    return results;
}
Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpty = 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` that 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 `Integers` and applies an operation on each element of that list. In the following listing you use this `forEach` method combined with a lambda to print all the elements of the list.

Listing 3.3 Working with a `Consumer`

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
public <T> void forEach(List<T> list, Consumer<T> c) {
    for(T t: list) {
        c.accept(t);
    }
}
forEach(
         Arrays.asList(1,2,3,4,5),
        (Integer i) -> System.out.println(i)                1
       );
```

1 **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` that 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 maps information from an input object to an output (for example, extracting the weight of an apple or mapping a string to its length). In the listing that follows we show how you can use it to create a method `map` to transform a list of `String`s into a list of `Integer`s containing the length of each `String`.

Listing 3.4 Working with a `Function`

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
public <T, R> List<R> map(List<T> list, Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T t: list) {
        result.add(f.apply(t));
    }
    return result;
}
// [7, 2, 6]
List<Integer> l = map(
                      Arrays.asList("lambdas", "in", "action"),
                      (String s) -> s.length()                  1
              );
```

1 **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 be bound only to reference types. This is due to how generics are internally implemented.[13] 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 opposite approach (that is, converting a reference type into a corresponding primitive type) is called *unboxing*. Java also has an *autoboxing* mechanism 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. 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 also added 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 {
    boolean test(int t);
}
IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);                                          ❶
Predicate<Integer> oddNumbers = (Integer i) -> i % 2 != 0;
oddNumbers.test(1000);                                           ❷
```

❶ **true (no boxing)**
❷ **false (boxing)**

In general, the names of functional interfaces that have a specialization for the input type parameter are 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 summarizes the most commonly used functional interfaces available in the Java API and their function descriptors, along with their primitive specializations. Keep in mind that

[13] **Some other languages such as C# don't have this restriction. Other languages such as Scala have only reference types. We revisit this issue in chapter 20.**

these are only a starter kit and you can always create your own if needed (Quiz 3.7 invents `TriFunction` just for this purpose). It can also help to create your own interfaces when a domain-specific name will help with program comprehension and maintenance. Remember, the notation `(T, U) -> R` shows how to think about a function descriptor. The left side of the arrow is a list representing the types of the arguments, and the right side represents the type of the result. 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 added 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<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, DoubleToIntFunction, DoubleToLongFunction, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>`
`Supplier<T>`	`() -> T`	`BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier`
`UnaryOperator<T>`	`T -> T`	`IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator`
`BinaryOperator<T>`	`(T, T) -> T`	`IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator`
`BiPredicate<T, U>`	`(T, U) -> boolean`	
`BiConsumer<T, U>`	`(T, U) -> void`	`ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>`
`BiFunction<T, U, R>`	`(T, U) -> R`	`ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>`

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, signatures of a lambda expression)? You'll find most of the answers in table 3.2. 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 a single abstract method called** `applyAsInt` **representing a function descriptor** `(int, int) -> int`**.**
3) `Consumer<T>` **has a 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`**.**
5) `BiFunction<T, U, R>` **has a single abstract method called** `apply` **representing a function descriptor** `(T, U) -> R`**.**

To summarize 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	Matching functional interface
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`	`IntBinaryOperator`
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 allow for a checked exception to be thrown. You have two options if you need a the body of a lambda expression to throw an exception: define your own functional interface that declares the checked exception, or wrap the lambda body with a `try/catch` block.

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

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
BufferedReaderProcessor p = (BufferedReader br) -> br.readLine();
```

But 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'll see in the next chapter that the Streams API makes heavy use of the functional 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 explain some more advanced details: 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. There's no need to fully understand the next section right away, and you may wish to come back to it later and move on to section 3.6 about method references.

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. In order to have a more formal understanding of lambda expressions, you should know 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 (for example, a method parameter that 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 scenes when you use a lambda expression. Figure 3.4 summarizes the type-checking process for the following code:

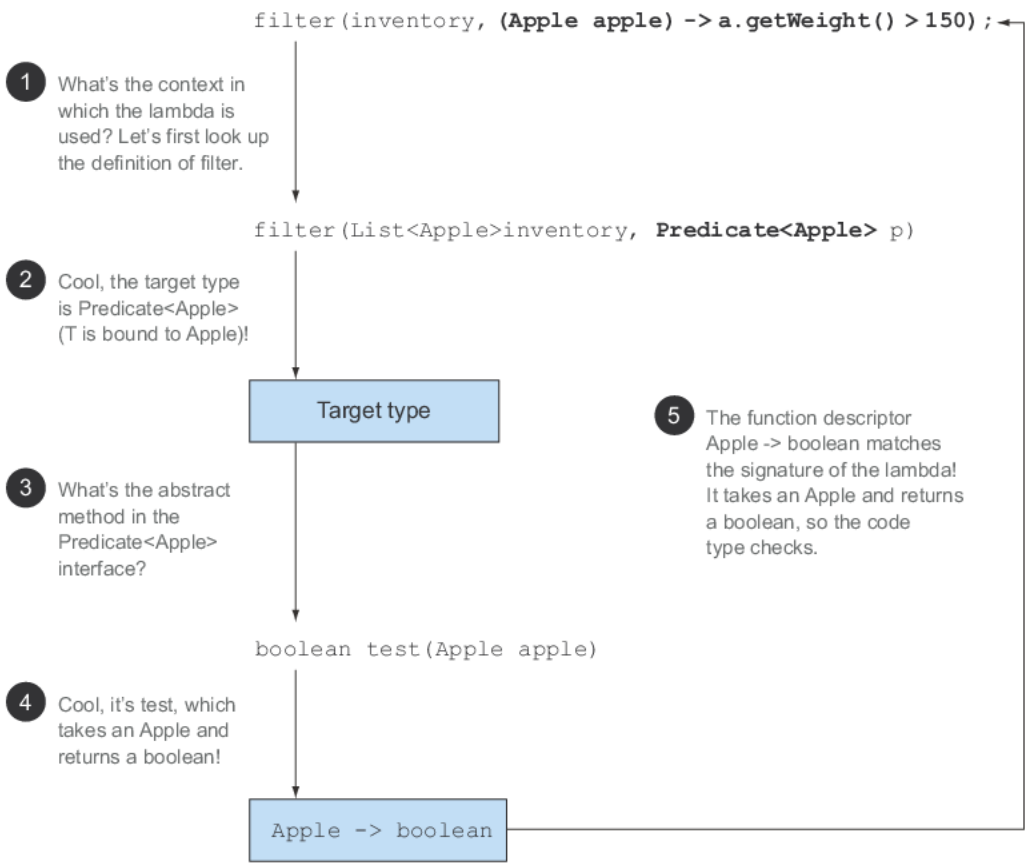

Figure 3.4 Deconstructing the type-checking process of a lambda expression

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

The type-checking process is deconstructed as follows:

- First, you look up the declaration of the `filter` method.
- Second, it expects as the second formal parameter an object of type `Predicate<Apple>` (the target type).
- Third, `Predicate<Apple>` is a functional interface defining a 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`. Note that if the lambda expression were throwing an exception, then the declared `throws` clause of the abstract method would also have to match.

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 functions that accept nothing and return a generic type `T`. The following two assignments are therefore valid:

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

Those of you who are familiar with Java's evolution will recall that Java 7 had already introduced the idea of types being inferred from context with generic inference using the diamond operator (`<>`) (this idea can be found 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 its body, it's compatible with a function descriptor that returns `void` (provided the parameter list is compatible too). 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 = (String s) -> list.add(s);
// Consumer has a void return
Consumer<String> b = (String 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 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"); };
```

Answer:

The context of the lambda expression is `Object` (the target type). But `Object` isn't a functional interface. To fix this you can change the target type to `Runnable`, which represents a function descriptor `() -> void`:

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

You could also fix the problem by casting the lambda expression to `Runnable` which explicitly provides a target type.

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

This technique can be useful in the context of overloading with a method taking two different functional interfaces that have the same function descriptor. You can cast the lambda in order to explicitly disambiguate which method signature should be selected.

For example, the call `execute(() -> {})` using the method `execute` below would be ambiguous as both `Runnable` and `Action` have the same function descriptor:

```
public void execute(Runnable runnable) {
    runnable.run();
}
public void execute(Action<T> action) {
    action.act();
}
@FunctionalInterface
interface Action {
    void act();
}
```

However, you can explicitly disambiguate the call by using a cast expression: `execute((Action) () -> {});`

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. The Java compiler deduces what functional interface to associate with a lambda expression from its surrounding context (the target type), meaning 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:[14]

```
List<Apple> greenApples =
        filter(inventory, apple -> GREEN.equals(apple.getColor()));  (1)
```

(1) 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());   (1)
Comparator<Apple> c =
  (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());               (2)
```

(1) Without type inference
(2) With type inference

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

3.5.4 Using local variables

All the lambda expressions we've shown so far used only their arguments inside their body. But lambda expressions are also allowed to use *free variables* (variables that aren't 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);
```

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

Nonetheless, there's a small twist: there are some restrictions on what you can do with these variables. Lambdas are allowed to capture (that is, to reference in their bodies) instance variables and static variables without restrictions. But when local variables are captured they have to be explicitly declared `final` or be 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;
Runnable r = () -> System.out.println(portNumber);          ❶
portNumber = 31337;
```

❶ **Error: local variables referenced from a lambda expression must be final or effectively final.**

RESTRICTIONS ON LOCAL VARIABLES

You may be asking yourself why local variables have these restrictions. First, there's a key difference in how instance and local variables are implemented behind the scenes. 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 that allocated the variable had deallocated it. Hence, Java implements access to a free local variable as access to a copy of it rather than access to the original variable. This makes no difference if the local variable is assigned to only once—hence the restriction.

Second, this restriction also discourages typical imperative programming patterns (which, as we explain in later chapters, prevent easy parallelization) that mutate an outer variable.

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 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 great feature that was introduced in Java 8 code: *method references*. Think of them 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 appear 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.Comparator.comparing`):

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

❶ **Your first method reference!**

Don't worry about the new syntax and how things work. You'll learn that over the next few sections!

3.6.1 In a nutshell

Why should you care about method references? Method references can be seen as shorthand for lambdas calling only 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 the method `getWeight` defined in the `Apple` class. (Remember that no brackets are needed after `getWeight` because you're not calling it at the moment, you're merely quoting its name.) This method reference is shorthand for the lambda expression `(Apple apple) -> apple.getWeight()`. Table 3.4 gives a few more examples of possible method references in Java 8.

Table 3.4 Examples of lambdas and method reference equivalents

Lambda	Method reference equivalent
`(Apple apple) -> apple.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`
`(String s) -> this.isValidName(s)`	`this::isValidName`

You can think of method references as syntactic sugar for lambdas that refer only to a single method because you 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 an existing object or expression* (for example, suppose you have a local variable `expensiveTransaction` that holds an object of type `Transaction`, which supports an instance method `getValue`; you can write `expensiveTransaction::getValue`)

The second and third kinds of method references may be a bit overwhelming at first. The idea with the second kind of method references such as `String::length` is that you're referring to a method to an object that will be supplied as one of the parameters of the lambda. For example, the lambda expression `(String s) -> s.toUpperCase()` can be rewritten as `String::toUpperCase`. But the third kind of method references refers to a situation when you're calling a method in a lambda to an external object that already exists. For example, the lambda expression `() -> expensiveTransaction.getValue()` can be rewritten as `expensiveTransaction::getValue`. This third kind of method reference is particularly useful when you need to pass around a method defined as a private helper. For example, say you defined a helper method `isValidName`:

```
private boolean isValidName(String string) {
    return Character.isUpperCase(string.charAt(0));
}
```

You can now pass this method around in the context of a `Predicate<String>` using a method reference:

```
filter(words, this::isValidName)
```

To help you digest this new knowledge, the shorthand rules to refactor a lambda expression to an equivalent method reference follow simple recipes, shown in figure 3.5.

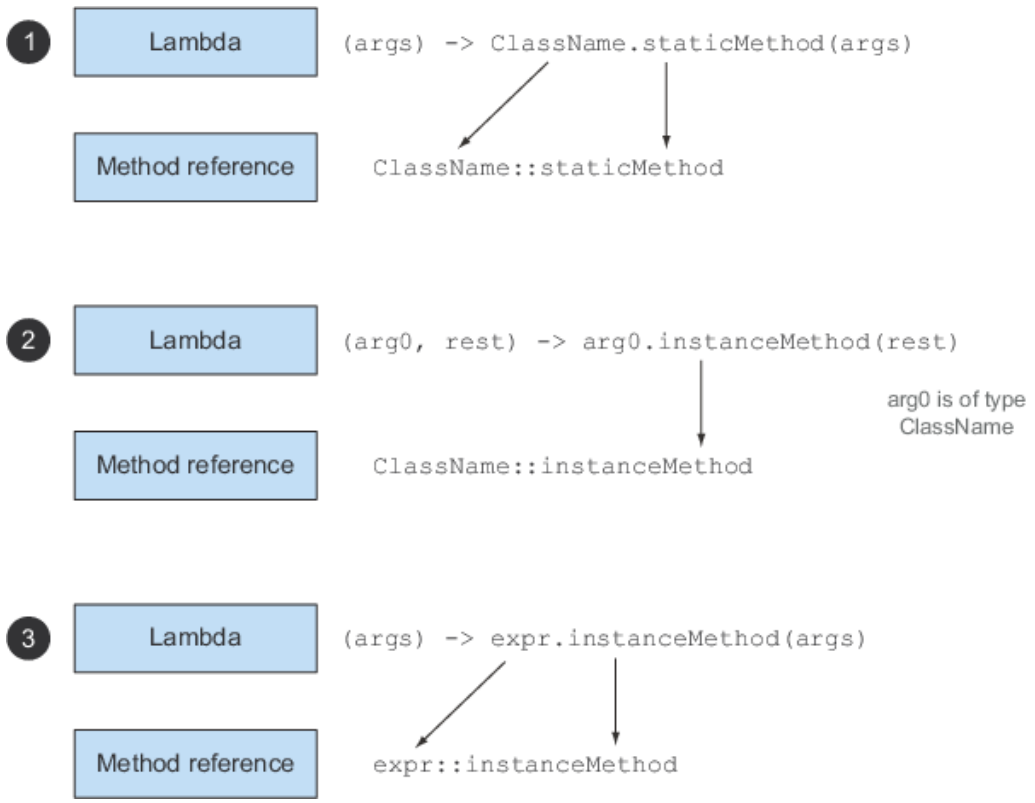

Figure 3.5 Recipes for constructing method references for three different types of lambda expressions

Note that there are also special forms of method references for constructors, array constructors, and super-calls. 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 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, which is more to the point:

```
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. ToIntFunction<String> stringToInt =
                (String s) -> Integer.parseInt(s);
2. BiPredicate<List<String>, String> contains =
                (list, element) -> list.contains(element);
3. Predicate<String> startsWithNumber = (String string) -> this.startsWithNumber(string);
```

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 `int`. As a result, the lambda can be rewritten using recipe ① from figure 3.5 (lambda expressions calling a static method) as follows:**

```
ToIntFunction<String> stringToInt = 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 from figure 3.5 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.

3) **This expression-style lambda invokes a private helper method. You can use recipe ③ from figure 3.5 as follows:**

```
Predicate<String> startsWithNumber = this::startsWithNumber
```

So far we showed only 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's a zero-argument constructor. This fits the signature `() -> Apple` of `Supplier`; you can do the following,

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

❶ **A constructor reference to the default Apple() constructor.**
❷ **Calling Supplier's get method will produce a new Apple.**

which is equivalent to

```
Supplier<Apple> c1 = () -> new Apple();          (1)
Apple a1 = c1.get();                             (2)
```

(1) A lambda expression creating an Apple with the default constructor.
(2) Calling Supplier's get method will produce a new Apple.

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

```
Function<Integer, Apple> c2 = Apple::new;        (1)
Apple a2 = c2.apply(110);                        (2)
```

(1) A constructor reference to Apple(Integer weight).
(2) Calling the Function's apply method with the requested weight will produce an Apple.

which is equivalent to

```
Function<Integer, Apple> c2 = (weight) -> new Apple(weight);     (1)
Apple a2 = c2.apply(110);                                        (2)
```

(1) A lambda expression creating an Apple with a requested weight.
(2) Calling the Function's apply method with the requested weight will produce a new Apple object.

In the following code, each element of a `List` of `Integer`s 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);              (1)
public List<Apple> map(List<Integer> list, Function<Integer, Apple> f) {
    List<Apple> result = new ArrayList<>();
    for(Integer i: list) {
        result.add(f.apply(i));
    }
    return result;
}
```

(1) Passing a constructor reference to the map method

If you have a two-argument constructor, `Apple(Color color, Integer weight)`, it fits the signature of the `BiFunction` interface, so you can do this,

```
BiFunction<Color, Integer, Apple> c3 = Apple::new;          (1)
Apple c3 = c3.apply(GREEN, 110);                            (2)
```

(1) A constructor reference to Apple(String color, Integer weight).
(2) Calling the BiFunction's apply method with the requested color and weight will produce a new Apple object.

which is equivalent to

```
BiFunction<String, Integer, Apple> c3 =
```

```
    (color, weight) -> new Apple(color, weight);          (1)
Apple c3 = c3.apply(GREEN, 110);                          (2)
```

(1) A lambda expression creating an Apple with a requested color and weight.
(2) Calling the BiFunction's apply method with the requested color and weight will produce a new Apple object.

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:

```
static 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())                   (1)
              .apply(weight);                             (2)
}
```

(1) You get a Function<Integer, Fruit> from the map.
(2) Calling the Function's apply() method with an Integer weight parameter will provide the requested Fruit.

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 constructor references. What would you need to do in order to use a constructor reference for a three-argument constructor such as `RGB(int, int, int)`?

Answer:

You saw that the syntax for a constructor reference is `ClassName::new`, so in this case it's `RGB::new`. But you need a functional interface that will match the signature of that constructor reference. Because 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, RGB> colorFactory = RGB::new;
```

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

3.7 Putting lambdas and method references into practice!

To wrap up this chapter and all we've discussed on lambdas, we continue with our initial problem of sorting a list of `Apple`s 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 so far in the book: behavior parameterization, anonymous classes, lambda expressions, and method references. The final solution we 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: Pass code

You're lucky; the Java 8 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 compare two `Apple`s! 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.

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: Use an 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: Use lambda expressions

But your current solution is still verbose. Java 8 introduced lambda expressions, which provide a lightweight syntax to achieve the same goal: *passing code*. You saw that a lambda expression 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 you'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 like this:

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

Can you make your code even more readable? `Comparator` includes a static helper method called `comparing` that takes a `Function` extracting a `Comparable` key and produces a `Comparator` object (we explain why interfaces can have static methods in chapter 13). It can be used as follows (note that you 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(apple -> apple.getWeight()));
```

3.7.4 Step 4: Use 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's shorter; it's 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 convenience 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 performs an `or` operation between the two predicates. Moreover, you can also compose functions such that the result of one becomes the input of another function. You may wonder

how it's 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'll introduce are called *default methods* (that is, they're not abstract methods). We explain them in detail in chapter 13. For now, just trust us and read chapter 13 later when you want to find out more about default methods and what you can do with them.

3.8.1 Composing Comparators

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

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

REVERSED ORDER

What if you 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 you can simply modify the previous example to sort the apples by decreasing weight by reusing the initial `Comparator`:

```
inventory.sort(comparing(Apple::getWeight).reversed());          (1)
```

(1) Sorting by decreasing weight

CHAINING COMPARATORS

This is all nice, but what if you 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 their 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`. You can solve the problem elegantly again:

```
inventory.sort(comparing(Apple::getWeight)
         .reversed()                                  (1)
         .thenComparing(Apple::getCountry));          (2)
```

(1) Sorting by decreasing weight
(2) Sorting further by country when two apples have same weight

3.8.2 Composing Predicates

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

```
Predicate<Apple> notRedApple = redApple.negate();  (1)
```

❶ Produces the negation of the existing Predicate object redApple

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(apple -> apple.getWeight() > 150);        ❶
```

❶ Chaining two predicates to produce another Predicate object

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

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

❶ Chaining three predicates to construct a more complex Predicate object

Why is this great? From simpler lambda expressions you can represent more complicated lambda expressions that still read like the problem statement! Note that the precedence of methods `and` and `or` in the above chain is from left to right – there is no equivalent of bracketing. So `a.or(b).and(c)` must be read as `(a || b) && c`. Similarly, `a.and(b).or(c)` must be read as as `(a && b) || c`.

3.8.3 Composing Functions

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

The method `andThen` returns a function that first applies a given function 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 2, you can combine them to create a function `h` that first increments a number and then multiplies the result by 2:

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

❶ In mathematics you'd write g(f(x)) or (g o f)(x).
❷ This returns 4.

You can also use the method `compose` similarly to first apply the function given as argument to `compose` and then apply the function to the result. For example, in the previous example using `compose`, it 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);    (1)
int result = h.apply(1);                        (2)
```

(1) In mathematics you'd write f(g(x)) or (f o g)(x).
(2) This returns 3.

Figure 3.6 illustrates the difference between `andThen` and `compose`.

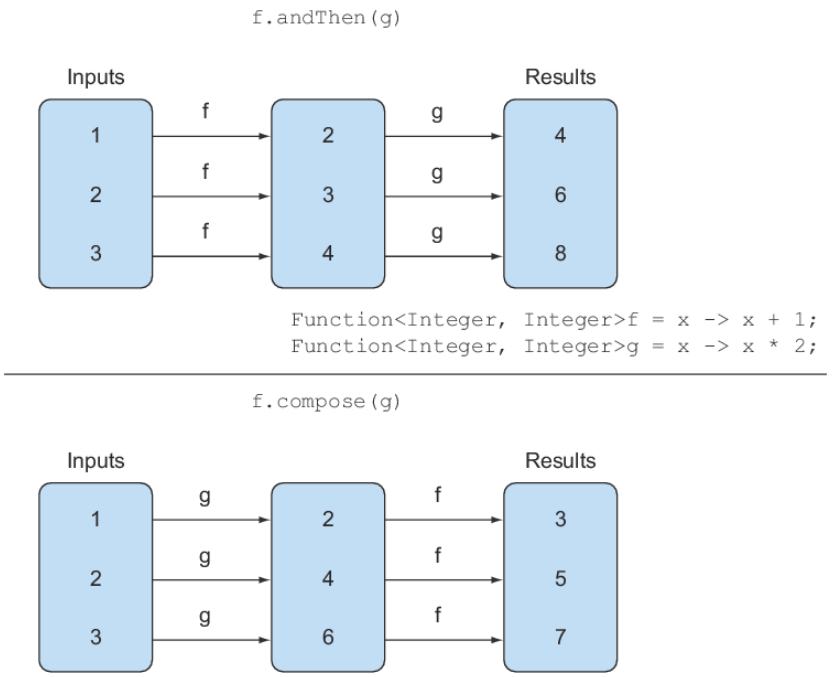

Figure 3.6 Using `andThen` vs. `compose`

This all sounds a 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);
```

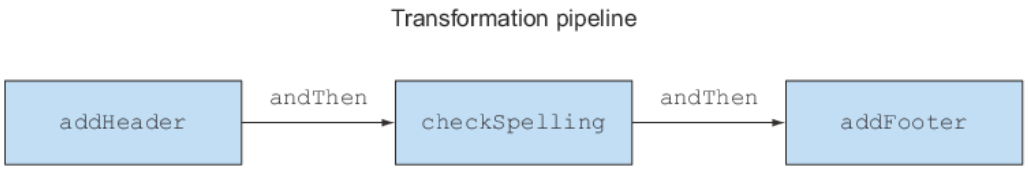

Figure 3.7 A transformation pipeline using `andThen`

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. Feel free to just skip it; nothing else in the book depends on it, but you may enjoy seeing another perspective.

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

$$\int_3^7 f(x)dx \quad or \quad \int_3^7 f(x+10)dx$$

for the area shown in figure 3.8.

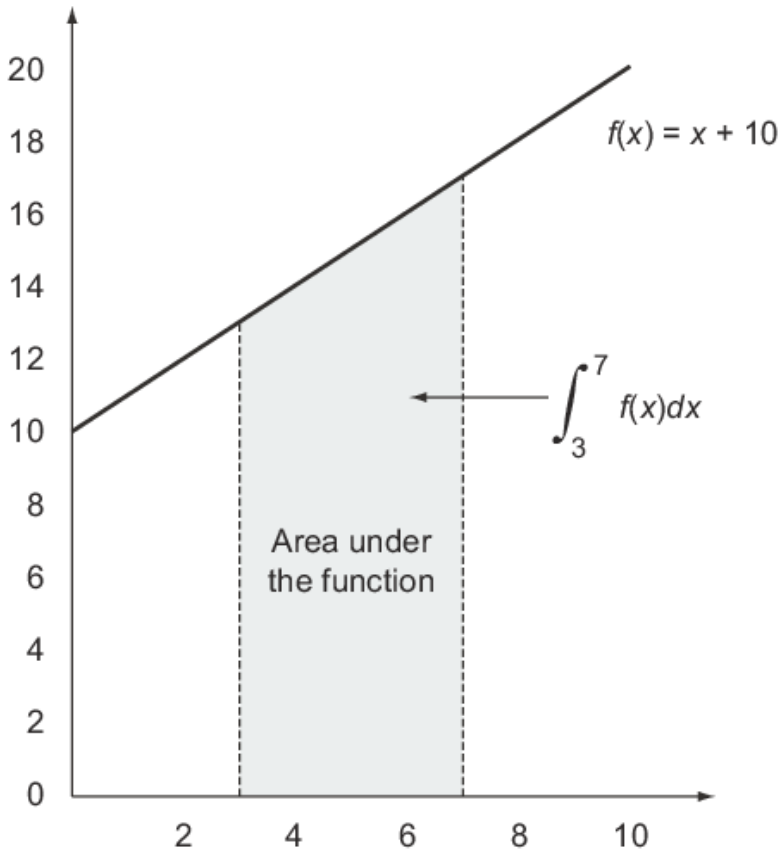

Figure 3.8 Area under the function `f(x) = x + 10` for `x` from 3 to 7

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 the solution:

$$1/2 \times ((3 + 10) + (7 + 10)) \times (7 - 3) = 60$$

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`, that 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 second, this would pass a value of `x+10` to integrate instead of passing the function `f`.

Indeed, the secret role of `dx` in 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 may now wonder how you'd write the method `integrate` itself. Continue to suppose 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) {    (1)
      return (f(a) + f(b)) * (b - a) / 2.0
}
```

(1) **Incorrect Java code! (You can't write functions as you do in mathematics.)**

But because lambda expressions can be used only in a context expecting a functional interface (in this case, `DoubleFunction`[15]), you have to write it this way:

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

[15] USing DoubleFunction<Double> is more efficient than using Function<Double,Double> as it avoids boxing the result

or using `DoubleUnaryOperator` which also avoids boxing the result:

```
public double integrate(DoubleUnaryOperator f, double a, double b) {
      return (f.applyAsDouble(a) + f.applyAsDouble(b)) * (b - a) / 2.0;
}
```

As a side remark, it's a bit of a shame that 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>`, described in table 3.2.
- 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 allocation and cleanup) can be used with lambdas to gain additional flexibility and reusability.
- The type expected for a lambda expression 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.