Chapter 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 Apples.
- ***An arrow—*** The arrow -> separates the list of parameters from the body of the lambda.
- ***The body of the lambda—*** Compare two Apples 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()
(Apple a) -> a.getWeight() > 150
(int x, int y) -> {
    System.out.println("Result:");
    System.out.println(x+y);
}
() -> 42
(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` (whether the apple is heavier than 150 g).

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.

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 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:

Only 4 and 5 are invalid lambdas.

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

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).

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 a 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> {
    int compare(T o1, T o2);
}
```

java.util.Comparator

```
public interface Runnable{
    void run();
}
```

java.lang.Runnable

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

java.awt.event.ActionListener

```
public interface Callable<V>{
    V call();
}
```

java.util.concurrent.Callable

```
public interface PrivilegedAction<V>{
    T run();
}
```

java.security.PrivilegedAction

Note

You'll see in chapter 9 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");

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

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

Using a lambda

Using an anonymous class

Prints "Hello World 1"

Prints "Hello World 2"

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).[1]

[1] 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.

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 Apples as parameters and returning an int. We'll provide more information about function descriptors in section 3.4 and table 3.2 later in the chapter.

Table 3.2. Common functional interfaces in Java 8

Functional interface	Function descriptor	Primitive specializations

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>, 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<L, R>	(L, R) -> 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 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 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.

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 15 and 16) 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). 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.

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!

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.

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

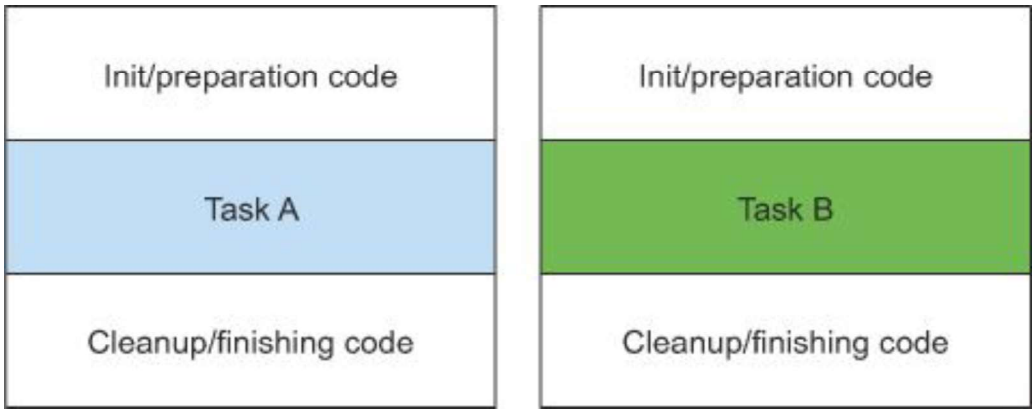
What about @FunctionalInterface?

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

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

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.

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

```
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 static 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

So far, we've showed 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 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> 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 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)
       );
```

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 Strings into a list of Integers containing the length of each String.

Listing 3.4. Working with a Function

```
@FunctionalInterface
public interface Function<T, R>{
    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<Integer> l = map(
                      Arrays.asList("lambdas","in","action"),
                      (String s) -> s.length()
                );
```

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.[2] 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):

[2] 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 16.

```
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 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{
    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, 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>, IntTo-DoubleFunction, and so on.

Table 3.2 gives a summary of the most commonly used functional interfaces available in the Java API and their function descriptors. Keep in mind that they're only a starter kit. You can always make your own if needed! Remember, the notation (T, U) -> R shows how to think about a function descriptor. The left side of the table 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.

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. Alternatively, Callable<T> also has 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 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 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 introduced a new functional interface Buffered-ReaderProcessor 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.

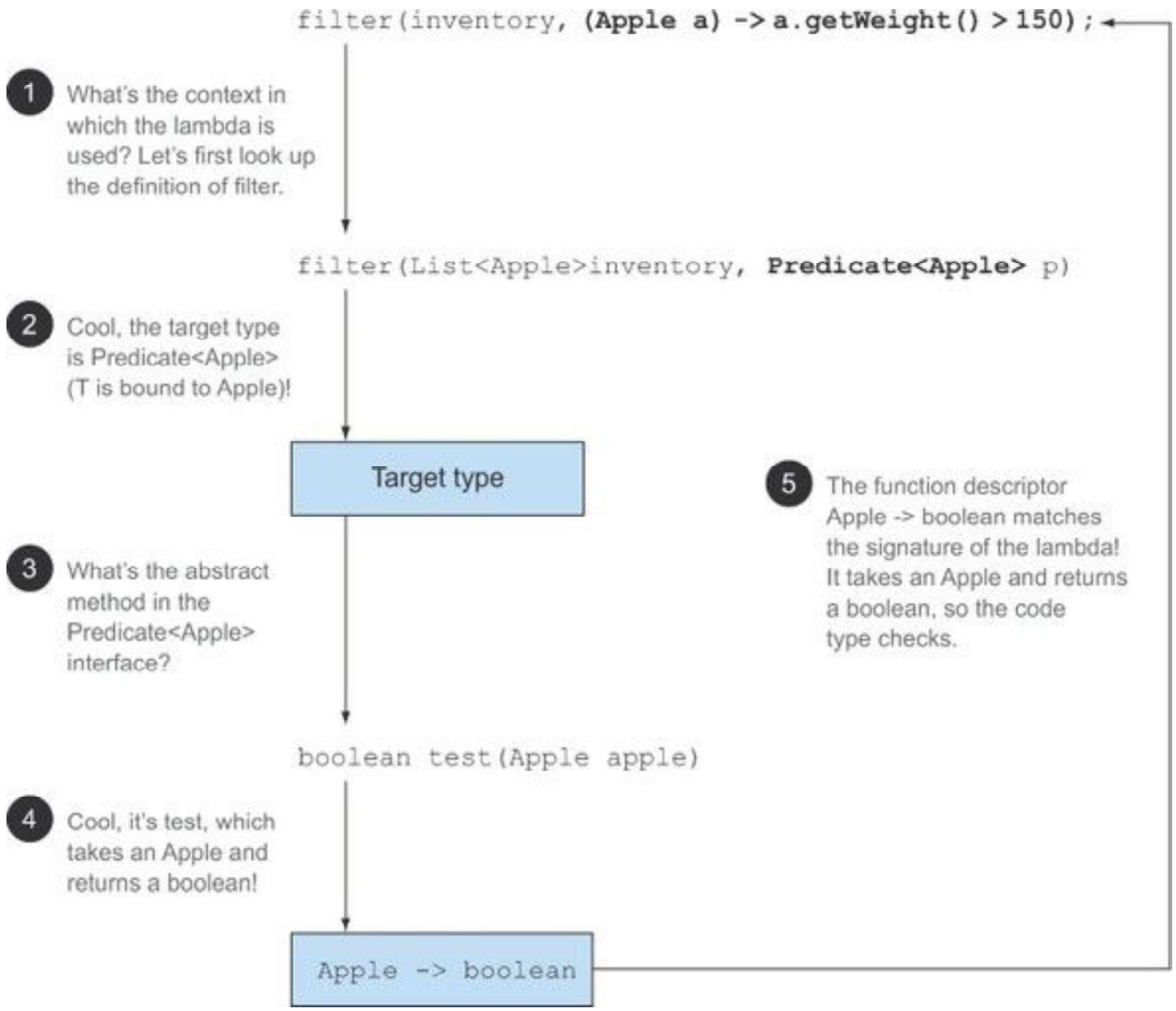
3.5. Type checking, type inference, and restrictions

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 a) -> a.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 = 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 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'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:[3]

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

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

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());
```

Without type inference

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

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);
```

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 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 them 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);
portNumber = 31337;
```

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 feature that you'll see 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));
```

Your first method reference!

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 because you're not actually calling the method. The method reference is shorthand for the lambda expression (Apple a) -> a.getWeight(). Table 3.4 gives a couple more examples of possible method references in Java 8.

Table 3.4. Examples of lambdas and method reference equivalents

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

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)

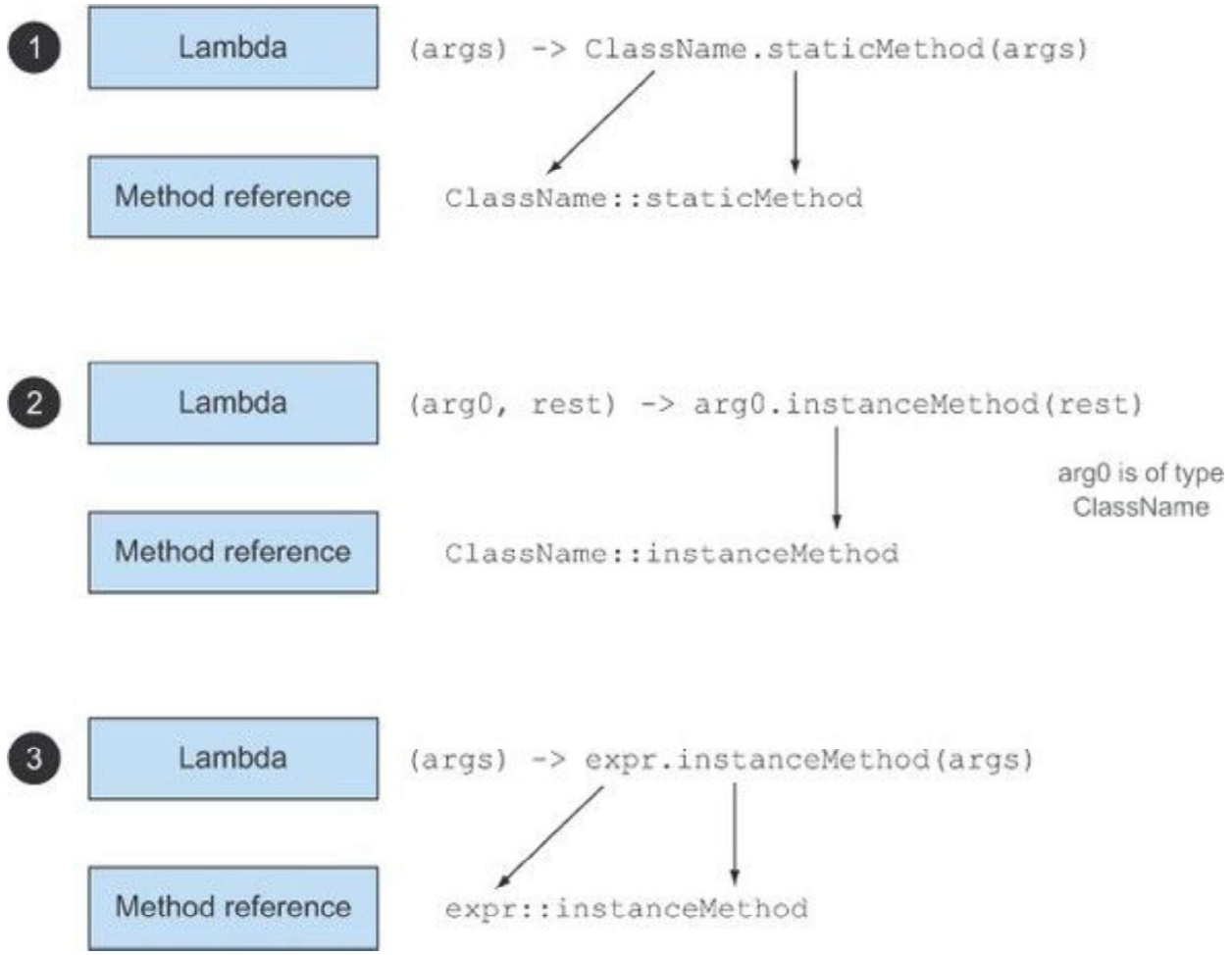
2. A method reference to an *instance method of an arbitrary type* (for example, the method length of a String, written String::length)

3. A method reference to an *instance method of an existing object* (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.

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:

```
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 ❶ from figure 3.5 (lambda expressions calling a static method) 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 ❷ 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.

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;
Apple a1 = c1.get();
```

A constructor reference to the default `Apple()` constructor.

Calling `Supplier`'s `get` method will produce a new `Apple`.

which is equivalent to

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

A lambda expression creating an `Apple` with the default constructor.

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;
Apple a2 = c2.apply(110);
```

A constructor reference to `Apple(Integer weight)`.

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);
Apple a2 = c2.apply(110);
```

A lambda expression creating an `Apple` with a requested weight.

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 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;
}
```

Passing a constructor reference to the map method

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

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

A constructor reference to Apple(String color, Integer weight).

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);
Apple c3 = c3.apply("green", 110);
```

A lambda expression creating an Apple with a requested color and weight.

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())
                .apply(weight);
}
```

You get a `Function<Integer, Fruit>` from the `map`.

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 Color(int, int, int)?

Answer:

You saw that the syntax for a constructor reference is ClassName::new, so in this case it's Color::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, Color> colorFactory = Color::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 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 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 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.

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 introduces 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 has 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 9). 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((a) -> a.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 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 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 9. For now, just trust us and read chapter 9 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());
```

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()
         .thenComparing(Apple::getCountry));
```

Sorting by decreasing weight

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();
```

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(a -> a.getWeight() > 150);
```

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(a -> a.getWeight() > 150)
            .or(a -> "green".equals(a.getColor()));
```

Chaining Predicate's methods 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 is managed from left to right using their positions in the chain. So a.or(b).and(c) can be seen 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);
int result = h.apply(1);
```

In math you'd write g(f(x)) or (g o f)(x).

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:

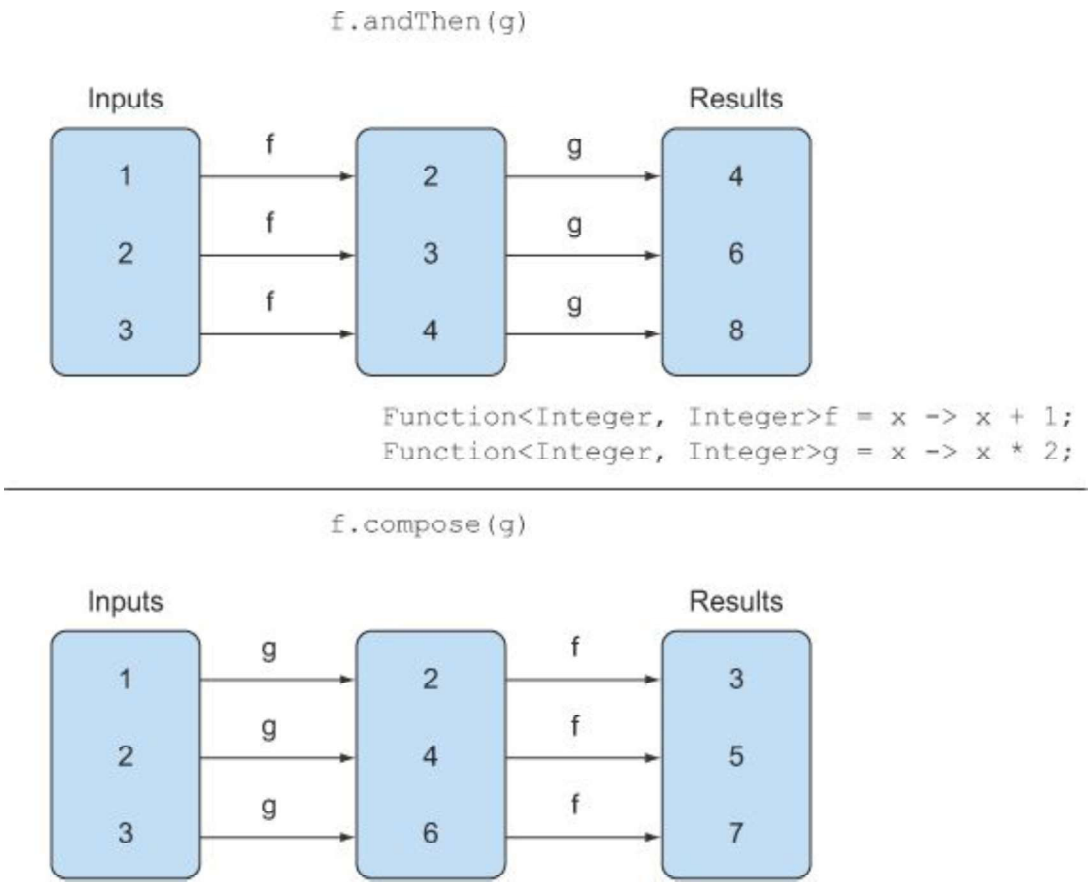
```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g);
int result = h.apply(1);
```

In math you'd write f(g(x)) or (f o g)(x).

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:

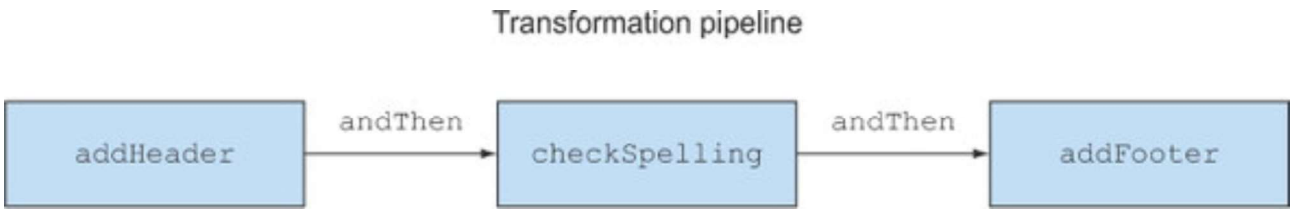
```
public class Letter{
    public static String addHeader(String text){
        return "From Raoul, Mario and Alan: " + text;
    }

    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:

Figure 3.7. A transformation pipeline using andThen




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

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

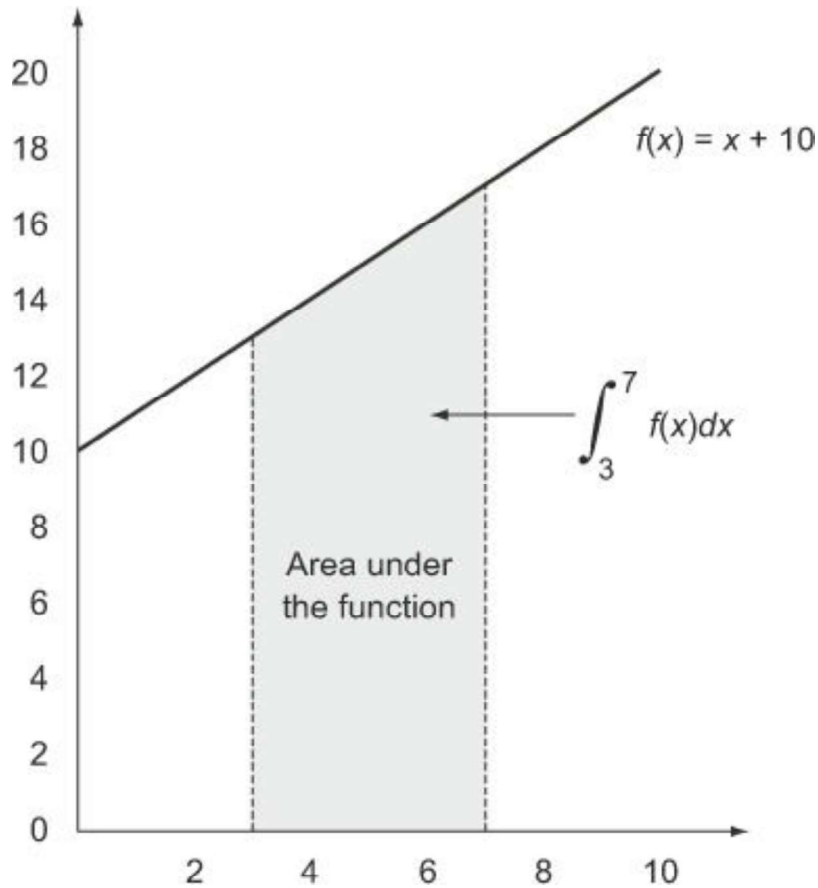
$f(x) = x + 10$

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

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

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 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) {
      return (f(a)+f(b))*(b-a)/2.0
}
```

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, Function), 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;
}

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.

Part 2. Functional-style data processing

The second part of this book is a deep exploration of the new Streams API, which lets you write powerful code that processes a collection of data in a declarative way. By the end of this second part, you'll have a full understanding of what streams are and how you can use them in your codebase to process a collection of data concisely and efficiently.

Chapter 4 introduces the concept of a stream and explains how it compares with a collection.

Chapter 5 investigates in detail the stream operations available to express sophisticated data processing queries. You'll look at many patterns such as filtering, slicing, finding, matching, mapping, and reducing.

Chapter 6 covers collectors—a feature of the Streams API that lets you express even more complex data processing queries.

In chapter 7, you'll learn about how streams can automatically run in parallel and leverage your multicore architectures. In addition, you'll learn about various pitfalls to avoid when using parallel streams correctly and effectively.

Chapter 4. Introducing streams

This chapter covers

- What is a stream?
- Collections vs. streams
- Internal vs. external iteration
- Intermediate vs. terminal operations

Collections is the most heavily used API in Java. What would you do without collections? Nearly every Java application *makes* and *processes* collections. Collections are fundamental to many programming tasks: they let you group and process data. To illustrate collections in action, imagine you want to create a collection of dishes to represent a menu and then iterate through it to sum the calories of each dish. You may want to process the collection to select only low-calorie dishes for a special healthy menu. But despite collections being necessary for almost any Java application, manipulating collections is far from perfect:

- Much business logic entails database-like operations such as *grouping* a list of dishes by category (for example, all vegetarian dishes) or *finding* the most expensive dish. How many times do you find yourself reimplementing these operations using iterators? Most databases let you specify such operations declaratively. For example, the following SQL query lets you select the names of dishes that are low in calories: SELECT name FROM dishes WHERE calorie < 400. As you can see, you don't need to implement how to filter using the attributes of a dish (for example, using an iterator and an accumulator). Instead, you express only what you expect. This basic idea means that you worry less about how to explicitly implement such queries—it's handled for you! Why can't you do something similar with collections?
- How would you process a large collection of elements? To gain performance you'd need to process it in parallel and leverage multicore architectures. But writing parallel code is complicated in comparison to working with iterators. In addition, it's no fun to debug!

So what could the Java language designers do to save your precious time and make your life easier as programmers? You may have guessed: the answer is *streams*.

4.1. What are streams?

Streams are an update to the Java API that lets you manipulate collections of data in a declarative way (you express a query rather than code an ad hoc implementation for it). For now

you can think of them as fancy iterators over a collection of data. In addition, streams can be processed in parallel *transparently*, without you having to write any multithreaded code! We explain in detail in chapter 7 how streams and parallelization work. Here's a taste of the benefits of using streams: compare the following code to return the names of dishes that are low in calories, sorted by number of calories, first in Java 7 and then in Java 8 using streams. Don't worry about the Java 8 code too much; we explain it in detail in the next sections!

Before (Java 7):

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: menu){
    if(d.getCalories() < 400){
        lowCaloricDishes.add(d);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(), d2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d: lowCaloricDishes){
    lowCaloricDishesName.add(d.getName());
}
```

Filter the elements using an accumulator.

Sort the dishes with an anonymous class.

Process the sorted list to select the names of dishes.

In this code you use a "garbage variable," lowCaloricDishes. Its only purpose is to act as an intermediate throwaway container. In Java 8, this implementation detail is pushed into the library where it belongs.

After (Java 8):

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
List<String> lowCaloricDishesName =
                menu.stream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dish::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

Select dishes that are below 400 calories.

Sort them by calories.

Extract the names of these dishes.

Store all the names in a List.

To exploit a multicore architecture and execute this code in parallel, you need only change stream() to parallelStream():

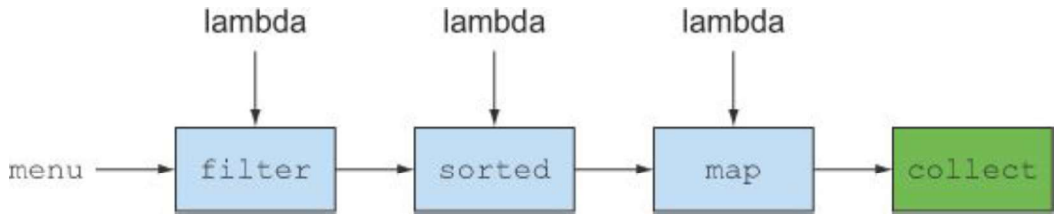
```
List<String> lowCaloricDishesName =
                    menu.parallelStream()
                        .filter(d -> d.getCalories() < 400)
```

```
                .sorted(comparing(Dishes::getCalories))
                .map(Dish::getName)
                .collect(toList());
```

You may be wondering what exactly happens when you call the method parallelStream. How many threads are being used? What are the performance benefits? Chapter 7 covers these questions in detail. For now, you can see that the new approach offers several immediate benefits from a software engineering point of view:

- The code is written in a *declarative way*: you specify *what* you want to achieve (that is, *filter* dishes that are *low* in calories) as opposed to specifying *how* to implement an operation (using control-flow blocks such as loops and if conditions). As you saw in the previous chapter, this approach, together with behavior parameterization, enables you to cope with changing requirements: you could easily create an additional version of your code to filter high-calorie dishes using a lambda expression, without having to copy and paste code.
- You chain together several building-block operations to express a complicated data processing pipeline (you chain the filter by linking sorted, map, and collect operations, as illustrated in figure 4.1) while keeping your code readable and its intent clear. The result of the filter is passed to the sorted method, which is then passed to the map method and then to the collect method.

Figure 4.1. Chaining stream operations forming a stream pipeline



Because operations such as filter (or sorted, map, and collect) are available as *high-level building blocks* that don't depend on a specific threading model, their internal implementation could be single-threaded or potentially maximize your multicore architecture transparently! In practice, this means you no longer have to worry about threads and locks to figure out how to parallelize certain data processing tasks: the Streams API does it for you!

The new Streams API is very expressive. For example, after reading this chapter and chapters 5 and 6, you'll be able to write code like this:

```
Map<Dish.Type, List<Dish>> dishesByType =
    menu.stream().collect(groupingBy(Dish::getType));
```

This particular example is explained in detail in chapter 6, “Collecting data with streams.” It basically groups dishes by their types inside a Map. For example, the Map may contain the following result:

```
{FISH=[prawns, salmon],
 OTHER=[french fries, rice, season fruit, pizza],
 MEAT=[pork, beef, chicken]}
```

Now try to think how you’d implement this with the typical imperative programming approach using loops. Don’t waste too much of your time. Embrace the power of streams in this and the following chapters!

Other libraries: Guava, Apache, and lambdaj

There have been many attempts at providing Java programmers with better libraries to manipulate collections. For example, Guava is a popular library created by Google. It provides additional container classes such as multimaps and multisets. The Apache Commons Collections library provides similar features. Finally, lambdaj, written by Mario Fusco, coauthor of this book, provides many utilities to manipulate collections in a declarative manner, inspired by functional programming.

Now Java 8 comes with its own official library for manipulating collections in a more declarative style.

To summarize, the Streams API in Java 8 lets you write code that’s

- ***Declarative—*** More concise and readable
- ***Composable—*** Greater flexibility
- ***Parallelizable—*** Better performance

For the remainder of this chapter and the next, we’ll use the following domain for our examples: a menu that’s nothing more than a list of dishes

```
List<Dish> menu = Arrays.asList(
    new Dish("pork", false, 800, Dish.Type.MEAT),
    new Dish("beef", false, 700, Dish.Type.MEAT),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
```

```
    new Dish("french fries", true, 530, Dish.Type.OTHER),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("season fruit", true, 120, Dish.Type.OTHER),
    new Dish("pizza", true, 550, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("salmon", false, 450, Dish.Type.FISH) );
```

where a Dish is an immutable class defined as

```
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;

    public Dish(String name, boolean vegetarian, int calories, Type type) {
        this.name = name;
        this.vegetarian = vegetarian;
        this.calories = calories;
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public int getCalories() {
        return calories;
    }

    public Type getType() {
        return type;
    }

    @Override
```

```
    public String toString() {
        return name;
    }

    public enum Type { MEAT, FISH, OTHER }
}
```

We'll now explore how you can use the Streams API in more detail. We'll compare streams to collections and provide some background. In the next chapter, we'll investigate in detail the stream operations available to express sophisticated data processing queries. We'll look at many patterns such as filtering, slicing, finding, matching, mapping, and reducing. There will be many quizzes and exercises to try to solidify your understanding.

Next, we'll discuss how you can create and manipulate numeric streams, for example, to generate a stream of even numbers or Pythagorean triples! Finally, we'll discuss how you can create streams from different sources such as from a file. We'll also discuss how to generate streams with an infinite number of elements—something you definitely can't do with collections!

4.2. Getting started with streams

We start our discussion of streams with collections, because that's the simplest way to begin working with streams. Collections in Java 8 support a new stream method that returns a stream (the interface definition is available in java.util.stream.Stream). You'll later see that you can also get streams in various other ways (for example, generating stream elements from a numeric range or from I/O resources).

So first, what exactly is a *stream*? A short definition is "a sequence of elements from a source that supports data processing operations." Let's break down this definition step by step:

- ***Sequence of elements—*** Like a collection, a stream provides an interface to a sequenced set of values of a specific element type. Because collections are data structures, they're mostly about storing and accessing elements with specific time/space complexities (for example, an ArrayList vs. a LinkedList). But streams are about expressing computations such as filter, sorted, and map that you saw earlier. Collections are about data; streams are about computations. We explain this idea in greater detail in the coming sections.
- ***Source—*** Streams consume from a data-providing source such as collections, arrays, or I/O resources. Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.

- ***Data processing operations—*** Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on. Stream operations can be executed either sequentially or in parallel.

In addition, stream operations have two important characteristics:

- ***Pipelining—*** Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline. This enables certain optimizations that we explain in the next chapter, such as *laziness* and *short-circuiting*. A pipeline of operations can be viewed as a database-like query on the data source.
- ***Internal iteration—*** In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration behind the scenes for you. We briefly mentioned this idea in chapter 1 and return to it later in the next section.

Let's look at a code example to explain all of these ideas:

```
import static java.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames =
  menu.stream()
      .filter(d -> d.getCalories() > 300)
      .map(Dish::getName)
      .limit(3)
      .collect(toList());
System.out.println(threeHighCaloricDishNames);
```

Get a stream from menu (the list of dishes).

Create a pipeline of operations: first filter high-calorie dishes.

Get the names of the dishes.

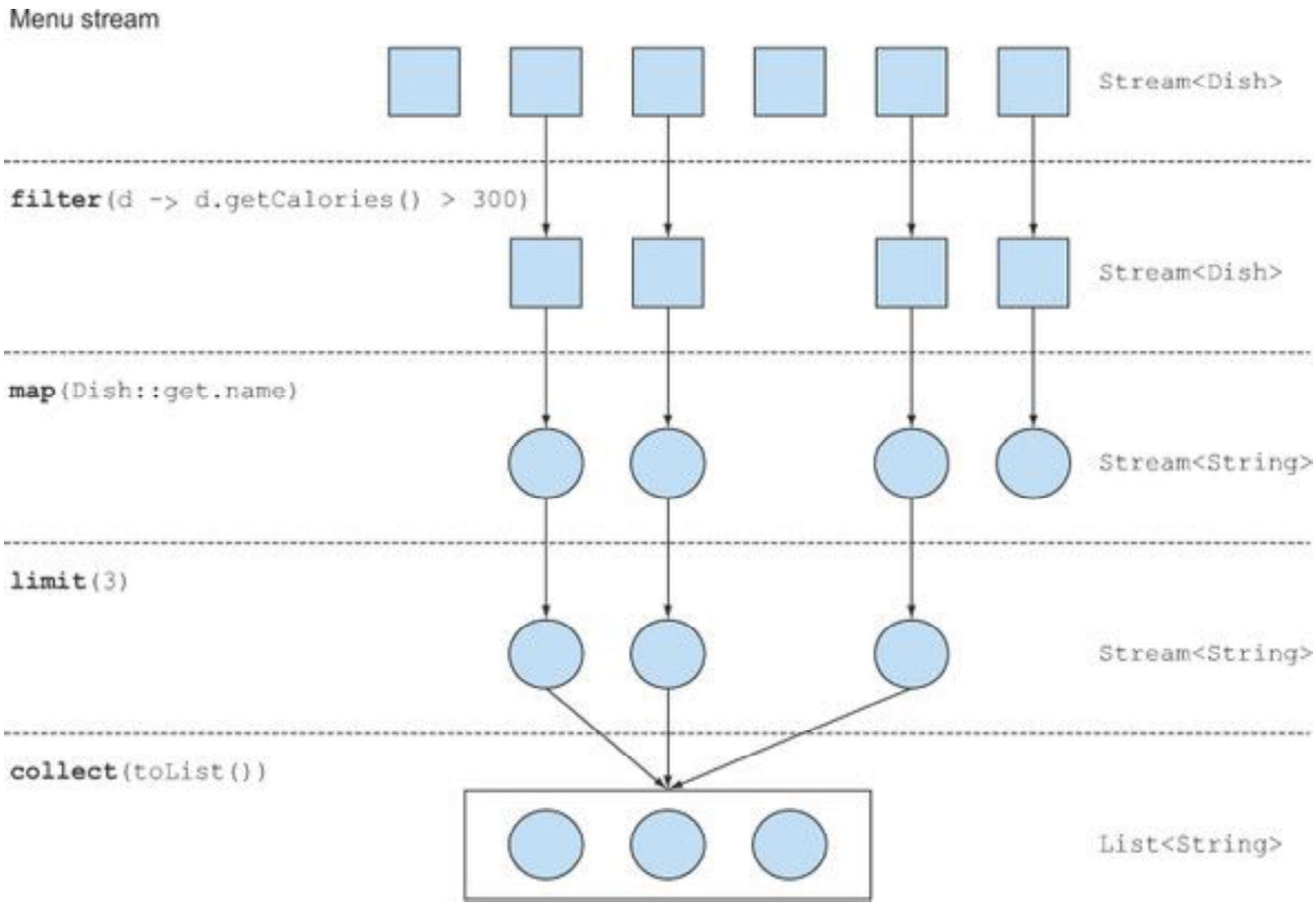
Select only the first three.

Store the results in another List.

The result is [pork, beef, chicken].

In this example, you first get a stream from the list of dishes by calling the stream method on menu. The *data source* is the list of dishes (the menu) and it provides *a sequence of elements* to the stream. Next, you apply a series of *data processing operations* on the stream: filter, map, limit, and collect. All these operations except collect return another stream so they can be connected to form a *pipeline*, which can be viewed as a query on the source. Finally, the collect operation starts processing the pipeline to return a result (it's different because it returns something other than a stream—here, a List). No result is produced, and indeed no element from menu is even selected, until collect is invoked. You can think of it as if the method invocations in the chain are queued up until collect is called. Figure 4.2 shows the sequence of stream operations: filter, map, limit, and collect, each of which is briefly described here:

Figure 4.2. Filtering a menu using a stream to find out three high-calorie dish names



- filter—Takes a lambda to exclude certain elements from the stream. In this case, you select dishes that have more than 300 calories by passing the lambda d -> d.getCalories() > 300.
- map—Takes a lambda to transform an element into another one or to extract information. In this case, you extract the name for each dish by passing the method reference Dish::getName, which is equivalent to the lambda d -> d.getName().
- limit—Truncates a stream to contain no more than a given number of elements.
- collect—Converts a stream into another form. In this case you convert the stream into a list. It looks like a bit of magic; we describe how collect works in more detail in chapter 6. At the moment, you can see collect as an operation that takes as an argument various recipes for accumulating the elements of a stream into a summary result. Here, toList() describes a recipe for converting a stream into a list.

Notice how the code we just described is very different than what you'd write if you were to process the list of menu items step by step. First, you use a much more declarative style to process the data in the menu where you say *what* needs to be done: "Find names of three

high-calorie dishes." You don't implement the filtering (filter), extracting (map), or truncating (limit) functionalities; they're available through the Streams library. As a result, the Streams API has more flexibility to decide how to optimize this pipeline. For example, the filtering, extracting, and truncating steps could be merged into a single pass and stop as soon as three dishes are found. We show an example to demonstrate that in the next chapter.

Let's now stand back a little and examine the conceptual differences between the Collections API and the new Streams API, before we explore in more detail what operations you can perform with a stream.

4.3. 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 of frames—we don't care which here) because it contains the whole data structure. Now consider watching the same video when it's being *streamed* over the internet. This is now a stream (of bytes or frames). The streaming video player needs to have downloaded only a few frames in advance of where the user is watching, so you can start displaying values from the beginning of the stream before most of the values in the stream have even been computed (consider streaming a live football game). 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 you had to wait for the final frame to appear before you could start showing the video. You 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 that holds *all* the values 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.)

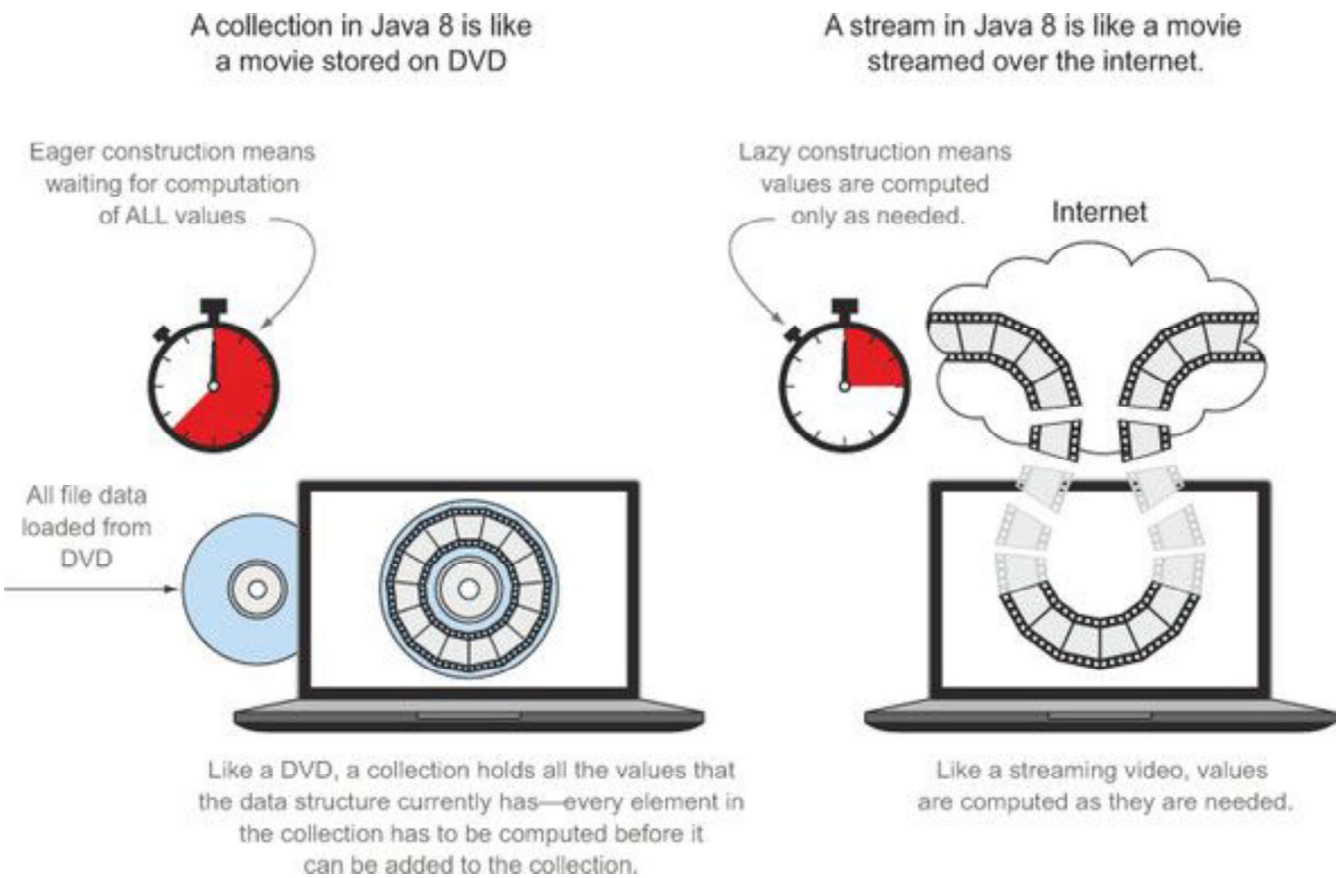
By contrast, a stream is a conceptually fixed data structure (you can't add or remove elements from it) whose elements are *computed on demand*. This gives rise to significant programming benefits. In chapter 6 we show how simple it is to construct a stream containing all the prime

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 produced—invisibly to the user—only *as* and *when* required. This is a form of a producer-consumer relationship. Another view is that a stream is like a lazily constructed collection: values are computed when they're 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.

Figure 4.3 illustrates the difference between a stream and a collection applied to our DVD vs. internet streaming example.

Figure 4.3. Streams vs. collections



Another example is a browser internet search. Suppose you search for a phrase with many matches in Google or in an e-commerce online shop. Instead of waiting for the whole collection

of results along with their photographs to be downloaded, you get a stream whose elements are the best 10 or 20 matches, along with a button to click for the next 10 or 20. When you, the consumer, click for the next 10, the supplier computes these on demand, before returning them to your browser for display.

4.3.1. Traversable only once

Note that, similarly to iterators, a stream can be traversed only once. After that a stream is said to be consumed. You can get a new stream from the initial data source to traverse it again just like for an iterator (assuming it's a repeatable source like a collection; if it's an I/O channel, you're out of luck). For example, the following code would throw an exception indicating the stream has been consumed:

```
List<String> title = Arrays.asList("Java8", "In", "Action");
Stream<String> s = title.stream();
s.forEach(System.out::println);
s.forEach(System.out::println);
```

Prints each word in the title.

java.lang.IllegalStateException: stream has already been operated upon or closed.

So keep in mind that you can consume a stream only once!

Streams and collections philosophically

For readers who like philosophical viewpoints, you can see a stream as a set of values spread out in time. In contrast, a collection is a set of values spread out in space (here, computer memory), which all exist at a single point in time—and which you access using an iterator to access members inside a for-each loop.

Another key difference between collections and streams is how they manage the iteration over data.

4.3.2. External vs. internal iteration

Using the Collection interface requires iteration to be done by the user (for example, using for-each); this is called *external iteration*. The Streams library by contrast uses *internal 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. The following code listings illustrate this difference.

Listing 4.1. Collections: external iteration with a for-each loop

```
List<String> names = new ArrayList<>();
for(Dish d: menu){
    names.add(d.getName());
}
```

Explicitly iterate the list of menu sequentially.

Extract the name and add it to an accumulator.

Note that the for-each hides some of the iteration complexity. The for-each construct is syntactic sugar that translates into something much uglier using an Iterator object.

Listing 4.2. Collections: external iteration using an iterator behind the scenes

```
List<String> names = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish d = iterator.next();
    names.add(d.getName());
}
```

Iterating explicitly

Listing 4.3. Streams: internal iteration

```
List<String> names = menu.stream()
                     .map(Dish::getName)
                     .collect(toList());
```

Start executing the pipeline of operations; no iteration!

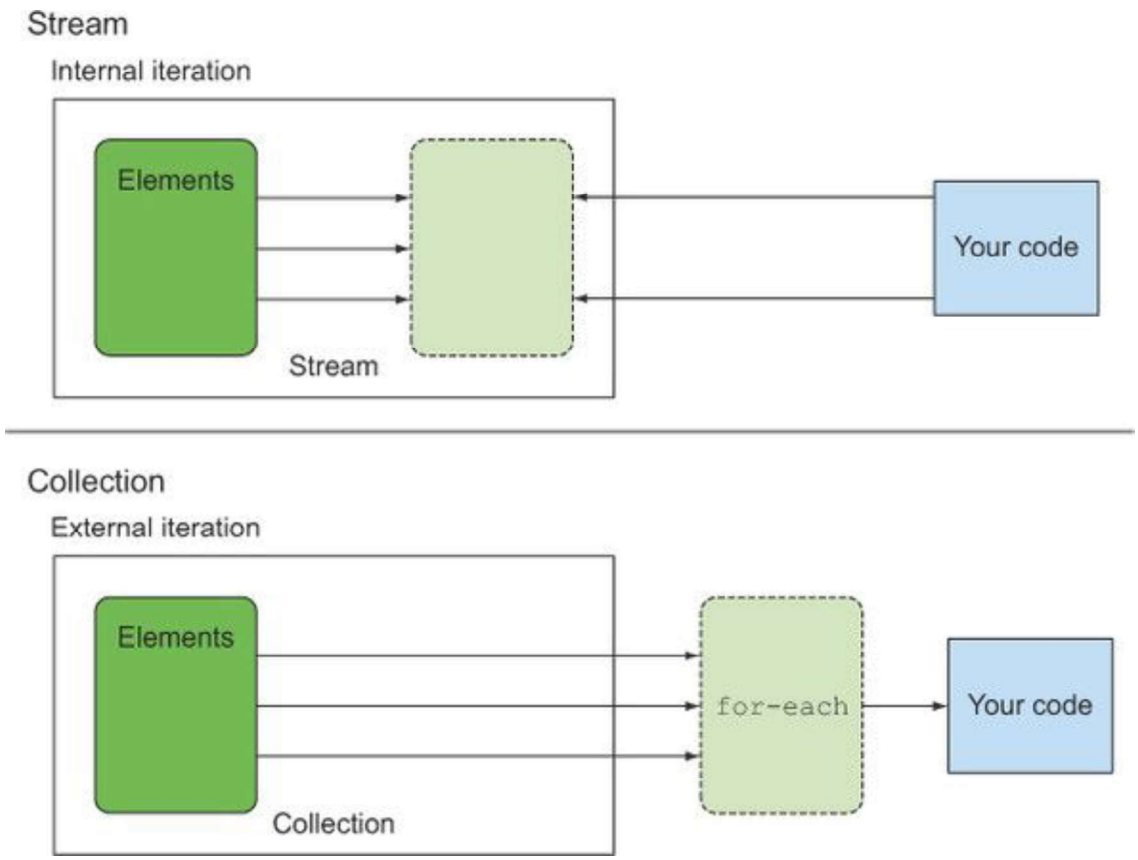
Parameterize `map` with the `getName` method to extract the name of a dish.

Let's use an analogy to understand the differences and benefits of internal iteration. Let's say you're 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 every day with your Java collections. You iterate a collection *externally*, explicitly pulling out and processing the items one by one. It would be far better if you could just tell Sofia, "Put all the toys that are on the floor 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 difficult if you iterate the collection externally as you're used to doing in Java. 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 Streams 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 for-each, then you've essentially committed to self-manage any parallelism. (*Self-managing* in practice means either "one fine day we'll parallelize this" or "starting the long and arduous battle involving tasks and synchronized".) Java 8 needed an interface like Collection but without iterators, ergo Stream! Figure 4.4 illustrates the difference between a stream (internal iteration) and a collection (external iteration).

Figure 4.4. Internal vs. external iteration



We've described the conceptual differences between collections and streams. Specifically, streams make use of internal iteration: iteration is taken care of for you. But this is useful only if you have a list of predefined operations to work with (for example, filter or map) that hide the iteration. Most of these operations take lambda expressions as arguments so you can parameterize their behavior as we showed in the previous chapter. The Java language designers shipped the Streams API with an extensive list of operations you can use to express complicated data processing queries. We'll briefly look at this list of operations now and explore them in more detail with examples in the next chapter.

4.4. Stream operations

The Stream interface in java.util.stream.Stream defines many operations. They can be classified into two categories. Let's look at our previous example once again:

```
List<String> names = menu.stream()
                     .filter(d -> d.getCalories() > 300)
                     .map(Dish::getName)
                     .limit(3)
                     .collect(toList());
```

Get a stream from the list of dishes.

Intermediate operation.

Intermediate operation.

Intermediate operation.

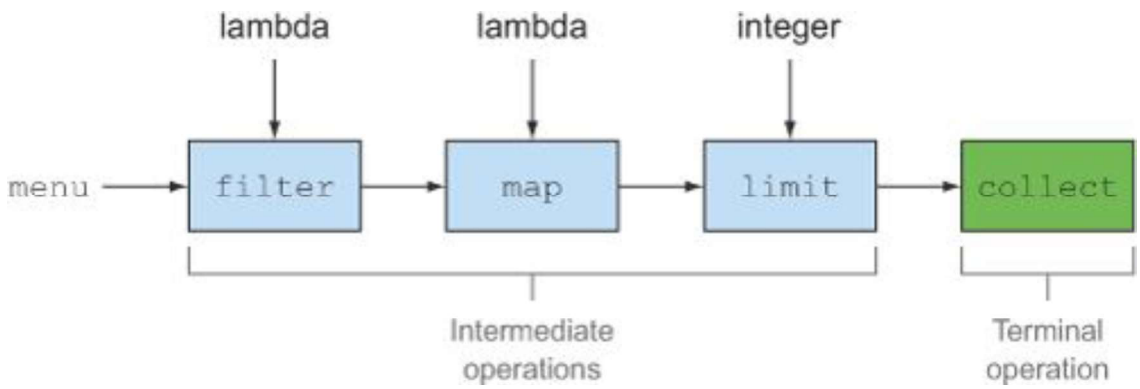
Converts the Stream into a List.

You can see two groups of operations:

- filter, map, and limit can be connected together to form a pipeline.
- collect causes the pipeline to be executed and closes it.

Stream operations that can be connected are called *intermediate operations*, and operations that close a stream are called *terminal operations*. Figure 4.5 highlights these two groups. So why is the distinction important?

Figure 4.5. Intermediate vs. terminal operations



4.4.1. Intermediate operations

Intermediate operations such as filter or sorted return another stream as the return type. This allows the operations to be connected to form a query. What's important is that intermediate operations don't perform any processing until a terminal operation is invoked on the stream pipeline—they're lazy. This is because intermediate operations can usually be merged and processed into a single pass by the terminal operation.

To understand what's happening in the stream pipeline, modify the code so each lambda also prints the current dish it's processing (like many demonstration and debugging techniques, this is appalling programming style for production code but directly explains the order of evaluation when you're learning):

```
List<String> names =
    menu.stream()
        .filter(d -> {
                System.out.println("filtering" + d.getName());
                return d.getCalories() > 300;
            })
        .map(d -> {
                System.out.println("mapping" + d.getName());
                return d.getName();
            })
        .limit(3)
        .collect(toList());
System.out.println(names);
```

Printing the dishes as they're filtered

Printing the dishes as you extract their names

This code when executed will print the following:

filtering pork
mapping pork

```
filtering beef
mapping beef
filtering chicken
mapping chicken
[pork, beef, chicken]
```

You can notice several optimizations due to the lazy nature of streams. First, despite the fact that many dishes have more than 300 calories, only the first three are selected! This is because of the limit operation and a technique called *short-circuiting*, as we'll explain in the next chapter. Second, despite the fact that filter and map are two separate operations, they were merged into the same pass (we call this technique *loop fusion*).

4.4.2. Terminal operations

Terminal operations produce a result from a stream pipeline. A result is any nonstream value such as a List, an Integer, or even void. For example, in the following pipeline, forEach is a terminal operation that returns void and applies a lambda to each dish in the source. Passing System.out.println to forEach asks it to print every Dish in the stream created from menu:

```
menu.stream().forEach(System.out::println);
```

To check your understanding of intermediate versus terminal operations, try out Quiz 4.1.

Quiz 4.1: Intermediate vs. terminal operations

In the stream pipeline that follows, can you identify the intermediate and terminal operations?

```
long count = menu.stream()
                .filter(d -> d.getCalories() > 300)
                .distinct()
                .limit(3)
                .count();
```

Answer:

The last operation in the stream pipeline count returns a long, which is a non-Stream value. It's therefore a *terminal operation*. All previous operations, filter, distinct, limit, are connected and return a Stream. They are therefore *intermediate operations*.

4.4.3. Working with streams

To summarize, working with streams in general involves three items:

- A *data source* (such as a collection) to perform a query on
- A chain of *intermediate operations* that form a stream pipeline
- A *terminal operation* that executes the stream pipeline and produces a result

The idea behind a stream pipeline is similar to the builder pattern.[1] In the builder pattern, there's a chain of calls to set up a configuration (for streams this is a chain of intermediate operations), followed by a call to a build method (for streams this is a terminal operation).

[1] See http://en.wikipedia.org/wiki/Builder_pattern.

For convenience, tables 4.1 and 4.2 summarize the intermediate and terminal stream operations you've seen in the code examples so far. Note that this is an incomplete list of operations provided by the Streams API; you'll see several more in the next chapter!

Table 4.1. Intermediate operations

Operation	Type	Return type	Argument of the operation	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T, R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Intermediate	Stream<T>		

Table 4.2. Terminal operations

Operation	Type	Purpose
forEach	Terminal	Consumes each element from a stream and applies a lambda to each of them. The operation returns void.
count	Terminal	Returns the number of elements in a stream. The operation returns a long.
collect	Terminal	Reduces the stream to create a collection such as a List, a Map, or even an Integer. See chapter 6 for more detail.

In the next chapter, we detail the available stream operations with use cases so you can see what kinds of queries you can express with them. We look at many patterns such as filtering, slicing, finding, matching, mapping, and reducing, which can be used to express sophisticated data processing queries.

Because chapter 6 deals with collectors in great detail, the only use this chapter and the next one make of the collect() terminal operation on streams is the special case of collect(toList()), which creates a List whose elements are the same as those of the stream it's applied to.

4.5. Summary

Here are some key concepts to take away from this chapter:

- A stream is a sequence of elements from a source that supports data processing operations.
- Streams make use of internal iteration: the iteration is abstracted away through operations such as filter, map, and sorted.
- There are two types of stream operations: intermediate and terminal operations.
- Intermediate operations such as filter and map return a stream and can be chained together. They're used to set up a pipeline of operations but don't produce any result.
- Terminal operations such as forEach and count return a nonstream value and process a stream pipeline to return a result.
- The elements of a stream are computed on demand.

Chapter 5. Working with streams

This chapter covers

- Filtering, slicing, and matching
- Finding, matching, and reducing
- Using numeric streams such as ranges of numbers
- Creating streams from multiple sources
- Infinite streams

In the previous chapter, you saw that streams let you move from *external iteration* to *internal iteration*. Instead of writing code as follows where you explicitly manage the iteration over a collection of data (external iteration),

```
List<Dish> vegetarianDishes = new ArrayList<>();
for(Dish d: menu){
    if(d.isVegetarian()){
        vegetarianDishes.add(d);
    }
}
```

you can use the Streams API (internal iteration), which supports the filter and collect operations, to manage the iteration over the collection of data for you. All you need to do is pass the filtering behavior as argument to the filter method:

```
import static java.util.stream.Collectors.toList;
List<Dish> vegetarianDishes =
    menu.stream()
        .filter(Dish::isVegetarian)
        .collect(toList());
```

This different way of working with data is useful because you let the Streams API manage how to process the data. As a consequence, the Streams API can work out several optimizations behind the scenes. In addition, using internal iteration, the Streams API can decide to run your code in parallel. Using external iteration, this isn't possible because you're committed to a single-threaded step-by-step sequential iteration.

In this chapter, you'll have an extensive look at the various operations supported by the Streams API. These operations will let you express complex data processing queries such as filtering, slicing, mapping, finding, matching, and reducing. Next, we'll explore special cases of streams: numeric streams, streams built from multiple sources such as files and arrays, and finally infinite streams.

5.1. Filtering and slicing

In this section, we look at how to select elements of a stream: filtering with a predicate, filtering only unique elements, ignoring the first few elements of a stream, or truncating a stream to a given size.

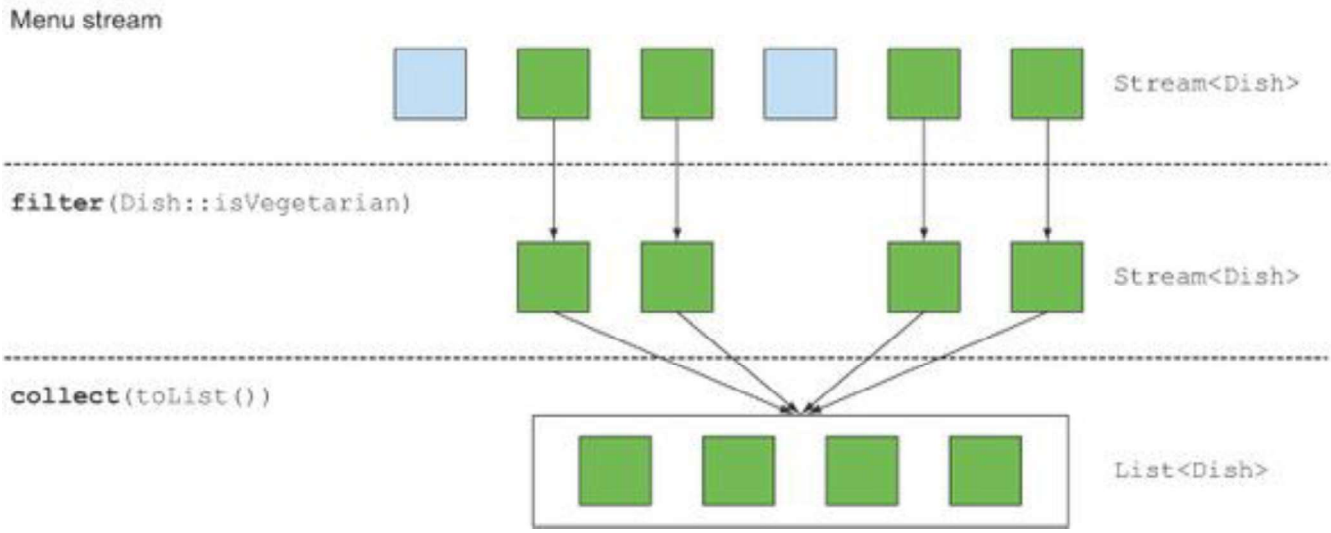
5.1.1. Filtering with a predicate

The Streams interface supports a filter method (which you should be familiar with by now). This operation takes as argument a *predicate* (a function returning a boolean) and returns a stream including all elements that match the predicate. For example, you can create a vegetarian menu by filtering all vegetarian dishes as follows and as illustrated in figure 5.1:

```
List<Dish> vegetarianMenu = menu.stream()
                                .filter(Dish::isVegetarian)
                                .collect(toList());
```

A method reference to check if a dish is vegetarian friendly

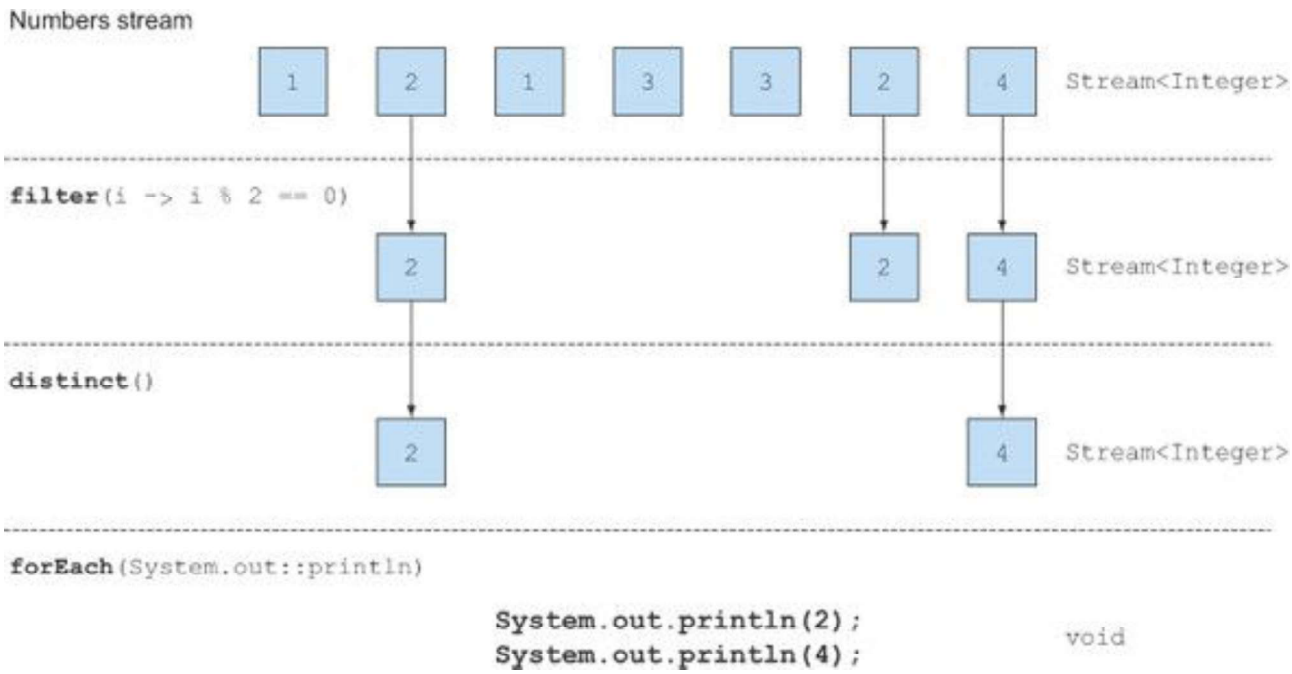
Figure 5.1. Filtering a stream with a predicate



5.1.2. Filtering unique elements

Streams also support a method called distinct that returns a stream with unique elements (according to the implementation of the hashCode and equals methods of the objects produced by the stream). For example, the following code filters all even numbers from a list and makes sure that there are no duplicates. Figure 5.2 shows this visually:

Figure 5.2. Filtering unique elements in a stream




```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
        .filter(i -> i % 2 == 0)
        .distinct()
        .forEach(System.out::println);
```

5.1.3. Truncating a stream

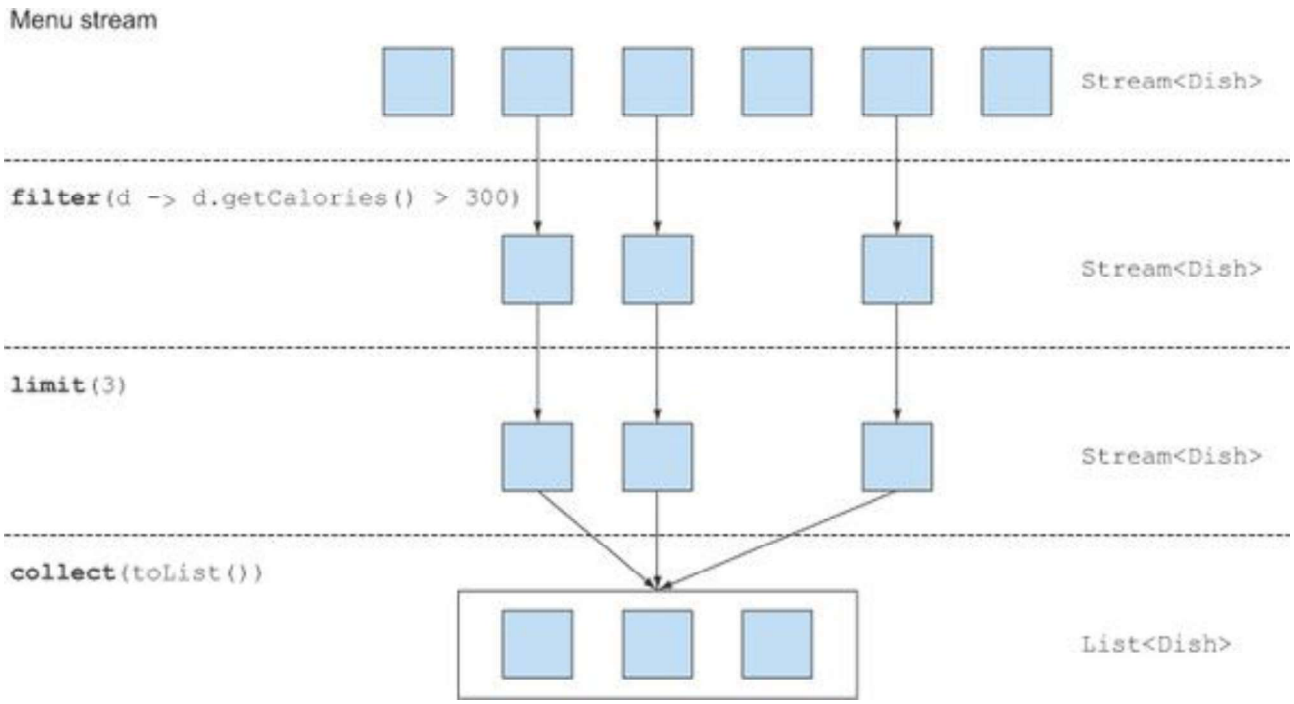
Streams support the limit(n) method, which returns another stream that's no longer than a given size. The requested size is passed as argument to limit. If the stream is ordered, the first elements are returned up to a maximum of n. For example, you can create a List by selecting the first three dishes that have more than 300 calories as follows:

```
List<Dish> dishes = menu.stream()
```

```
.filter(d -> d.getCalories() > 300)
.limit(3)
.collect(toList());
```

Figure 5.3 illustrates a combination of filter and limit. You can see that only the first three elements that match the predicate are selected and the result is immediately returned.

Figure 5.3. Truncating a stream

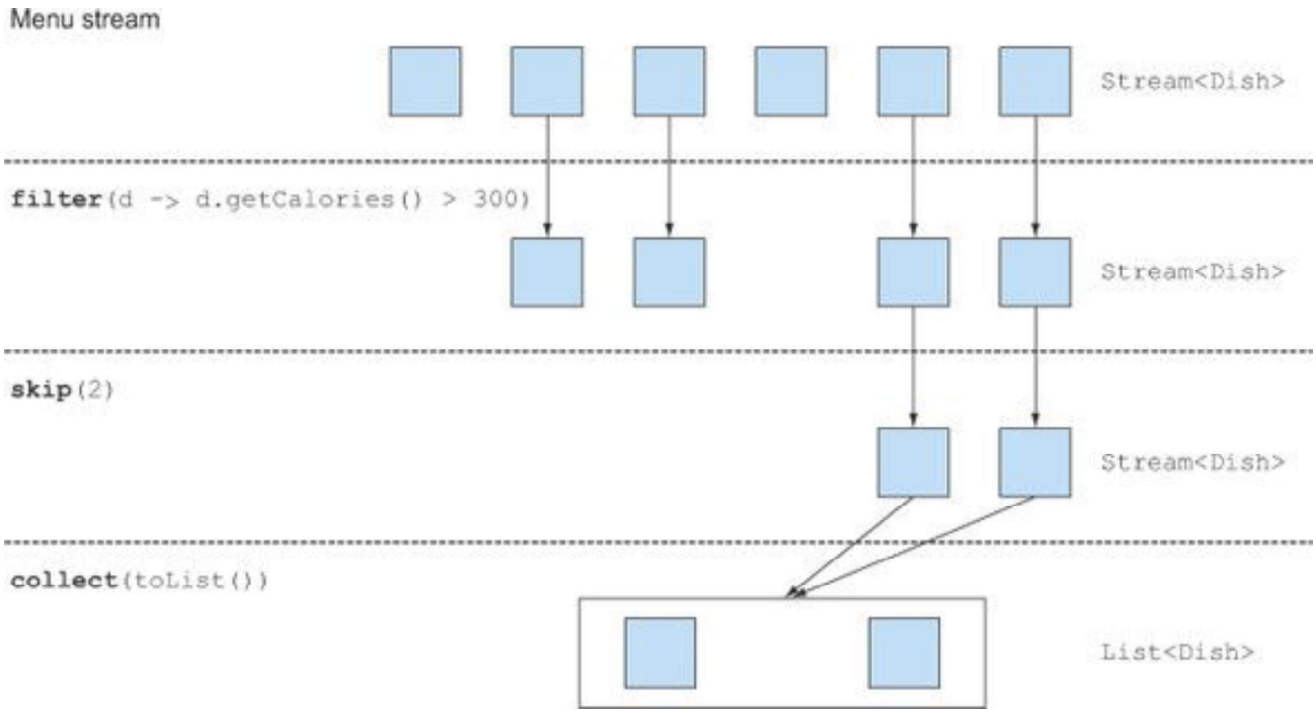


Note that limit also works on unordered streams (for example, if the source is a Set). In this case you shouldn't assume any order on the result produced by limit.

5.1.4. Skipping elements

Streams support the skip(n) method to return a stream that discards the first n elements. If the stream has fewer elements than n, then an empty stream is returned. Note that limit(n) and skip(n) are complementary! For example, the following code skips the first two dishes that have more than 300 calories and returns the rest. Figure 5.4 illustrates this query:

Figure 5.4. Skipping elements in a stream




```
List<Dish> dishes = menu.stream()
                    .filter(d -> d.getCalories() > 300)
                    .skip(2)
                    .collect(toList());
```

Put what you've learned in this section into practice with Quiz 5.1 before we move to mapping operations.

Quiz 5.1: Filtering

How would you use streams to filter the first two meat dishes?

Answer:

You can solve this problem by composing the methods filter and limit together and using collect(toList()) to convert the stream into a list as follows:

```
List<Dish> dishes =
    menu.stream()
        .filter(d -> d.getType() == Dish.Type.MEAT)
```

```
        .limit(2)
        .collect(toList());
```

5.2. Mapping

A very common data processing idiom is to select information from certain objects. For example, in SQL you can select a particular column from a table. The Streams API provides similar facilities through the map and flatMap methods.

5.2.1. Applying a function to each element of a stream

Streams support the method map, which takes a function as argument. The function is applied to each element, mapping it into a new element (the word *mapping* is used because it has a meaning similar to *transforming* but with the nuance of "creating a new version of" rather than "modifying"). For example, in the following code you pass a method reference Dish::getName to the map method to *extract* the names of the dishes in the stream:

```
List<String> dishNames = menu.stream()
                                .map(Dish::getName)
                                .collect(toList());
```

Because the method getName returns a String, the stream outputted by the map method is of type Stream<String>.

Let's take a slightly different example to solidify your understanding of map. Given a list of words, you'd like to return a list of the number of characters for each word. How would you do it? You'd need to apply a function to each element of the list. This sounds like a job for the map method! The function to apply should take a word and return its length. You can solve this problem as follows by passing a method reference String::length to map:

```
List<String> words = Arrays.asList("Java8", "Lambdas", "In", "Action");
List<Integer> wordLengths = words.stream()
                                .map(String::length)
                                .collect(toList());
```

Let's now return to the example where you extracted the name of each dish. What if you wanted to find out the length of the name of each dish? You could do this by chaining another map as follows:

```
List<Integer> dishNameLengths = menu.stream()
                                .map(Dish::getName)
                                .map(String::length)
                                .collect(toList());
```

5.2.2. Flattening streams

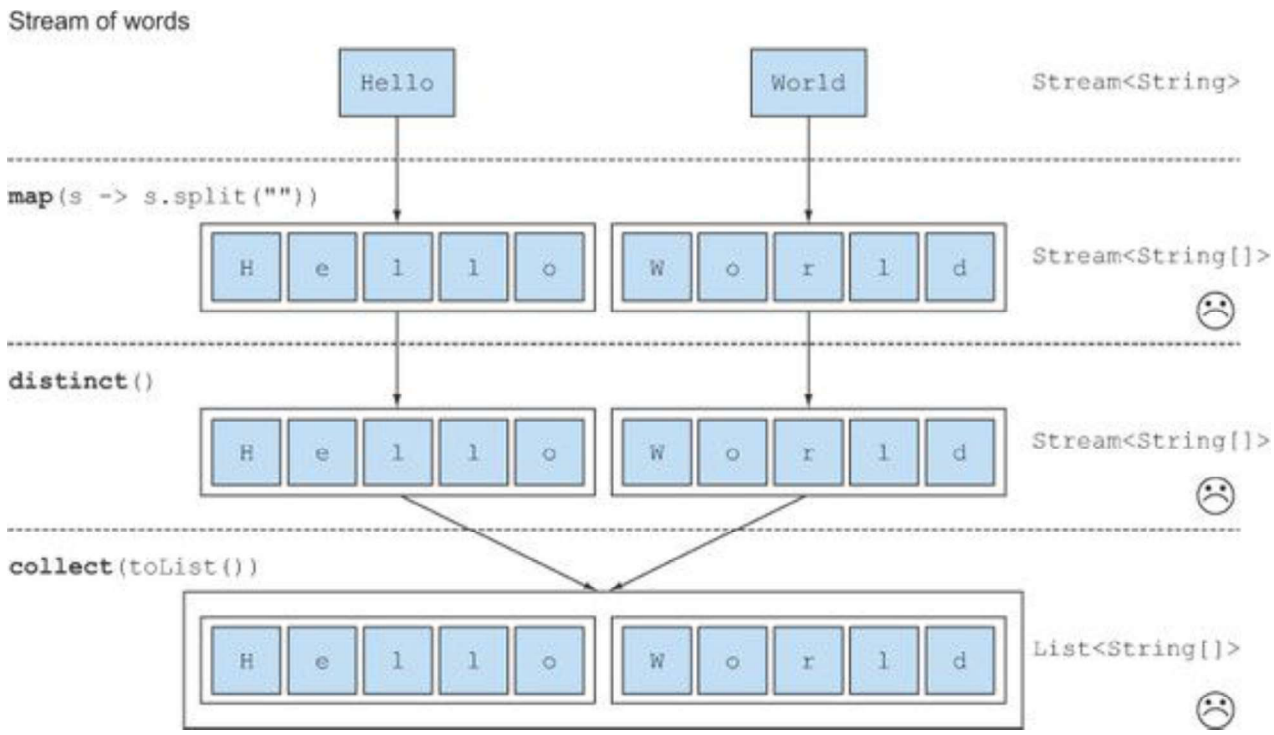
You saw how to return the length for each word in a list using the method map. Let's extend this idea a bit further: how could you return a list of all the *unique characters* for a list of words? For example, given the list of words ["Hello", "World"] you'd like to return the list ["H", "e", "l", "o", "W", "r", "d"].

You might think that this is easy, that you can just map each word into a list of characters and then call distinct to filter duplicate characters. A first go could be like this:

```
words.stream()
    .map(word -> word.split(""))
    .distinct()
    .collect(toList());
```

The problem with this approach is that the lambda passed to the map method returns a String[] (an array of String) for each word. So the stream returned by the map method is actually of type Stream<String[]>. What you really want is Stream<String> to represent a stream of characters. Figure 5.5 illustrates the problem.

Figure 5.5. Incorrect use of map to find unique characters from a list of words



Luckily there's a solution to this problem using the method flatMap! Let's see step by step how to solve it.

Attempt using map and Arrays.stream

First, you need a stream of characters instead of a stream of arrays. There's a method called Arrays.stream()that takes an array and produces a stream, for example:

```
String[] arrayOfWords = {"Goodbye", "World"};
Stream<String> streamOfwords = Arrays.stream(arrayOfWords);
```

Use it in the previous pipeline to see what happens:

```
words.stream()
      .map(word -> word.split(""))
      .map(Arrays::stream)
      .distinct()
      .collect(toList());
```

Converts each word into an array of its individual letters

Makes each array into a separate stream

The current solution still doesn't work! This is because you now end up with a list of streams (more precisely, Stream<Stream<String>>)! Indeed, you first convert each word into an array of its individual letters and then make each array into a separate stream.

Using flatMap

You can fix this problem by using flatMap as follows:

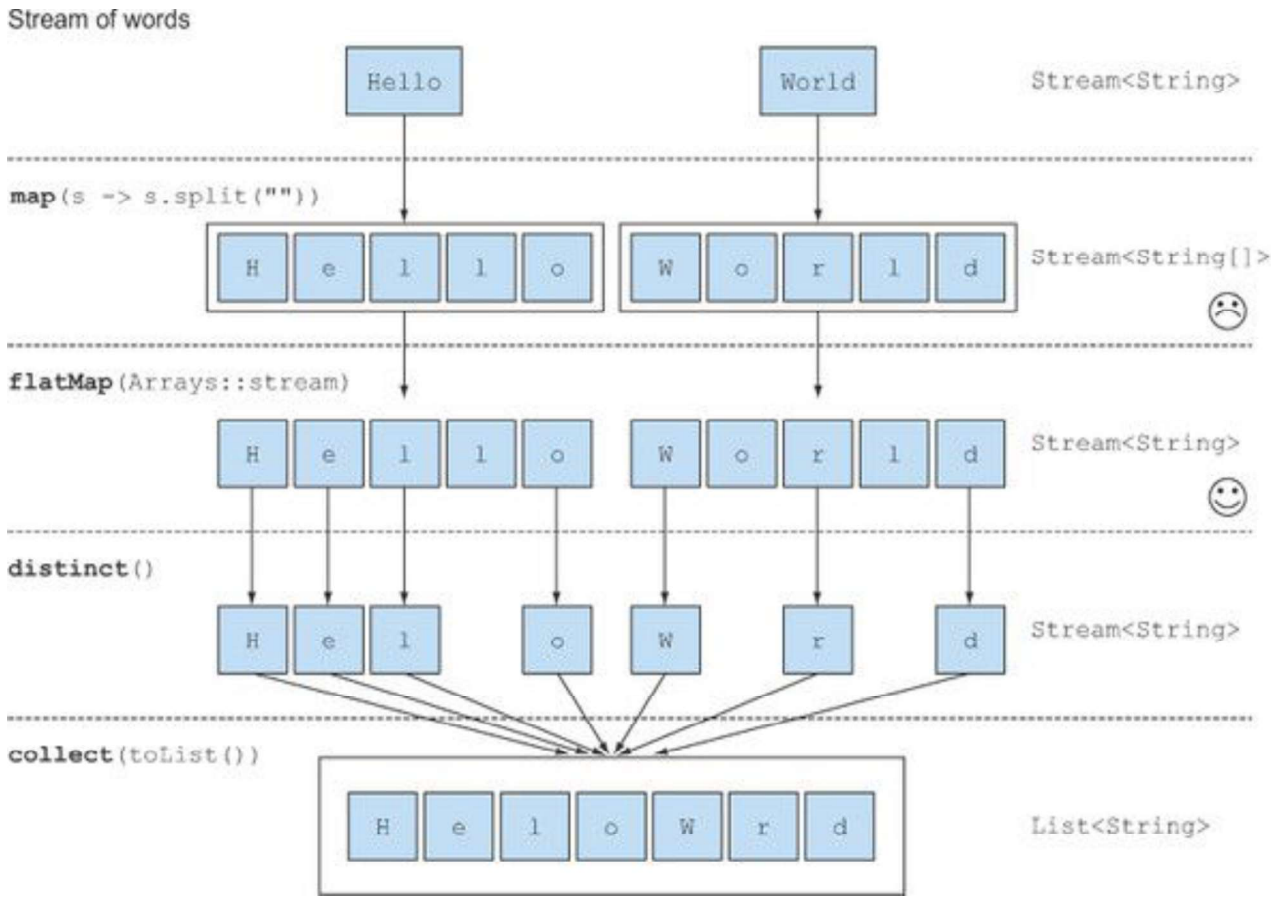
```
List<String> uniqueCharacters =
  words.stream()
       .map(w -> w.split(""))
       .flatMap(Arrays::stream)
       .distinct()
       .collect(Collectors.toList());
```

Converts each word into an array of its individual letters

Flattens each generated stream into a single stream

Using the flatMap method has the effect of mapping each array not with a stream but *with the contents of that stream*. All the separate streams that were generated when using map(Arrays::stream) get amalgamated—flattened into a single stream. Figure 5.6 illustrates the effect of using the flatMap method. Compare it with what map does in figure 5.5.

Figure 5.6. Using flatMap to find the unique characters from a list of words



In a nutshell, the flatMap method lets you replace each value of a stream with another stream and then concatenates all the generated streams into a single stream.

We come back to flatMap in chapter 10 when we discuss more advanced Java 8 patterns such as using the new library class Optional for null checking. To solidify your understanding of map and flatMap, try out Quiz 5.2.

Quiz 5.2: Mapping

1. Given a list of numbers, how would you return a list of the square of each number? For example, given [1, 2, 3, 4, 5] you should return [1, 4, 9, 16, 25].

Answer:

You can solve this problem by using map with a lambda that takes a number and returns the square of the number:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squares =
    numbers.stream()
            .map(n -> n * n)
            .collect(toList());
```

2. Given two lists of numbers, how would you return all pairs of numbers? For example, given a list [1, 2, 3] and a list [3, 4] you should return [(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]. For simplicity, you can represent a pair as an array with two elements.

Answer:

You could use two maps to iterate on the two lists and generate the pairs. But this would return a Stream<Stream<Integer[]>>. What you need to do is flatten the generated streams to result in a Stream<Integer[]>. This is what flatMap is for:

```
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<int[]> pairs =
    numbers1.stream()
            .flatMap(i -> numbers2.stream()
                                .map(j -> new int[]{i, j})
                    )
            .collect(toList());
```

3. How would you extend the previous example to return only pairs whose sum is divisible by 3? For example, (2, 4) and (3, 3) are valid.

Answer:

You saw earlier that filter can be used with a predicate to filter elements from a stream. Because after the flatMap operation you have a stream of int[] that represent a pair, you just need a predicate to check to see if the sum is divisible by 3:

```
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<int[]> pairs =
    numbers1.stream()
```

```
.flatMap(i ->
        numbers2.stream()
                .filter(j -> (i + j) % 3 == 0)
                .map(j -> new int[]{i, j})
    )
.collect(toList());
```

The result is [(2, 4), (3, 3)].

5.3. Finding and matching

Another common data processing idiom is finding whether some elements in a set of data match a given property. The Streams API provides such facilities through the allMatch, anyMatch, noneMatch, findFirst, and findAny methods of a stream.

5.3.1. Checking to see if a predicate matches at least one element

The anyMatch method can be used to answer the question “Is there an element in the stream matching the given predicate?” For example, you can use it to find out whether the menu has a vegetarian option:

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
    System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

The anyMatch method returns a boolean and is therefore a terminal operation.

5.3.2. Checking to see if a predicate matches all elements

The allMatch method works similarly to anyMatch but will check to see if all the elements of the stream match the given predicate. For example, you can use it to find out whether the menu is healthy (that is, all dishes are below 1000 calories):

```
boolean isHealthy = menu.stream()
                    .allMatch(d -> d.getCalories() < 1000);
```

noneMatch

The opposite of allMatch is noneMatch. It ensures that no elements in the stream match the given predicate. For example, you could rewrite the previous example as follows using noneMatch:

```
boolean isHealthy = menu.stream()
                                .noneMatch(d -> d.getCalories() >= 1000);
```

These three operations, anyMatch, allMatch, and noneMatch, make use of what we call *short-circuiting*, a stream version of the familiar Java short-circuiting && and || operators.

Short-circuiting evaluation

Some operations don't need to process the whole stream to produce a result. For example, say you need to evaluate a large boolean expression chained with and operators. You need only find out that one expression is false to deduce that the whole expression will return false, no matter how long the expression is; there's no need to evaluate the entire expression. This is what *short-circuiting* refers to.

In relation to streams, certain operations such as allMatch, noneMatch, findFirst, and findAny don't need to process the whole stream to produce a result. As soon as an element is found, a result can be produced. Similarly, limit is also a short-circuiting operation: the operation only needs to create a stream of a given size without processing all the elements in the stream. Such operations are useful, for example, when you need to deal with streams of infinite size, because they can turn an infinite stream into a stream of finite size. We show examples of infinite streams in section 5.7.

5.3.3. Finding an element

The findAny method returns an arbitrary element of the current stream. It can be used in conjunction with other stream operations. For example, you may wish to find a dish that's vegetarian. You can combine the filter method and findAny to express this query:

```
Optional<Dish> dish =
```

```
menu.stream()
    .filter(Dish::isVegetarian)
    .findAny();
```

The stream pipeline will be optimized behind the scenes to perform a single pass and finish as soon as a result is found by using short-circuiting. But wait a minute; what's this Optional thing in the code?

Optional in a nutshell

The Optional<T> class (java.util.Optional) is a container class to represent the existence or absence of a value. In the previous code, it's possible that findAny doesn't find any element. Instead of returning null, which is well known for being error prone, the Java 8 library designers introduced Optional<T>. We won't go into the details of Optional here, because we show in detail in chapter 10 how your code can benefit from using Optional to avoid bugs related to null checking. But for now, it's good to know that there are a few methods available in Optional that force you to explicitly check for the presence of a value or deal with the absence of a value:

- isPresent() returns true if Optional contains a value, false otherwise.
- ifPresent(Consumer<T> block) executes the given block if a value is present. We introduced the Consumer functional interface in chapter 3; it lets you pass a lambda that takes an argument of type T and returns void.
- T get() returns the value if present; otherwise it throws a NoSuchElement-Exception.
- T orElse(T other) returns the value if present; otherwise it returns a default value.

For example, in the previous code you'd need to explicitly check for the presence of a dish in the Optional object to access its name:

```
menu.stream()
    .filter(Dish::isVegetarian)
    .findAny()
    .ifPresent(d -> System.out.println(d.getName());
```

Returns an Optional<Dish>.

If a value is contained, it's printed; otherwise nothing happens.

5.3.4. Finding the first element

Some streams have an *encounter order* that specifies the order in which items logically appear in the stream (for example, a stream generated from a List or from a sorted sequence of data). For such streams you may wish to find the first element. There's the findFirst method for this, which works similarly to findAny. For example, the code that follows, given a list of numbers, finds the first square that's divisible by 3:

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree =
   someNumbers.stream()
                    .map(x -> x * x)
                    .filter(x -> x % 3 == 0)
                    .findFirst(); // 9
```

When to use findFirst and findAny

You may wonder why we have both findFirst and findAny. The answer is parallelism. Finding the first element is more constraining in parallel. If you don't care about which element is returned, use findAny because it's less constraining when using parallel streams.

5.4. Reducing

So far, the terminal operations you've seen return a boolean (allMatch and so on), void (forEach), or an Optional object (findAny and so on). You've also been using collect to combine all elements in a stream into a List.

In this section, you'll see how you can combine elements of a stream to express more complicated queries such as "Calculate the sum of all calories in the menu," or "What is the highest calorie dish in the menu?" using the reduce operation. Such queries combine all the elements in the stream repeatedly to produce a single value such as an Integer. These queries can be classified as *reduction operations* (a stream is reduced to a value). In functional programming-language jargon, this is referred to as a *fold* because you can view this operation as repeatedly folding a long piece of paper (your stream) until it forms a small square, which is the result of the fold operation.

5.4.1. Summing the elements

Before we investigate how to use the reduce method, it helps to first see how you'd sum the elements of a list of numbers using a for-each loop:

```
int sum = 0;
for (int x : numbers) {
      sum += x;
```

```
}
```

Each element of numbers is combined iteratively with the addition operator to form a result. You *reduce* the list of numbers into one number by repeatedly using addition. There are two parameters in this code:

- The initial value of the sum variable, in this case 0
- The operation to combine all the elements of the list, in this case +

Wouldn't it be great if you could also multiply all the numbers without having to repeatedly copy and paste this code? This is where the reduce operation, which abstracts over this pattern of repeated application, can help. You can sum all the elements of a stream as follows:

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

reduce takes two arguments:

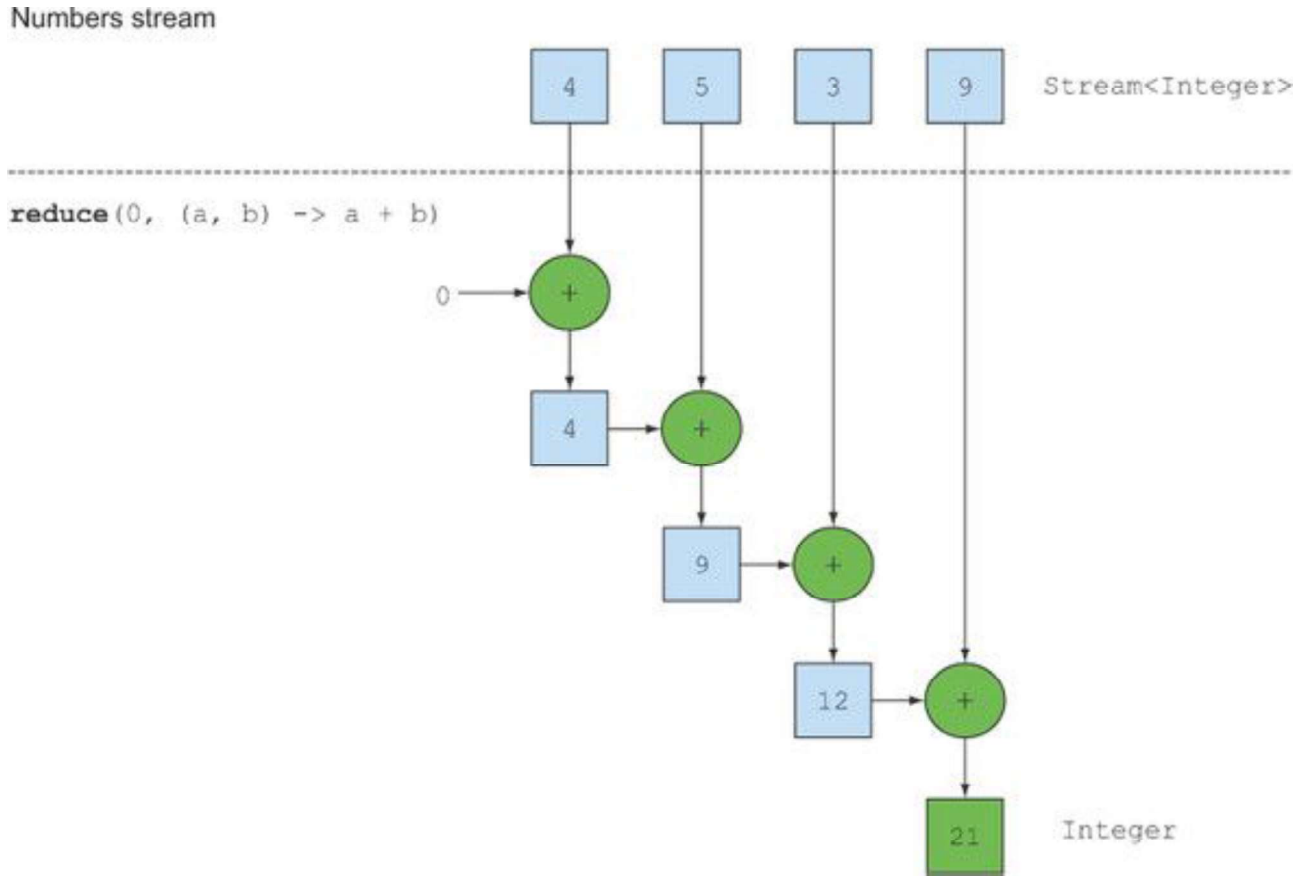
- An initial value, here 0.
- A BinaryOperator<T> to combine two elements and produce a new value; here you use the lambda (a, b) -> a + b.

You could just as easily multiply all the elements by passing a different lambda, (a, b) -> a * b, to the reduce operation:

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

Figure 5.7 illustrates how the reduce operation works on a stream: the lambda combines each element repeatedly until the stream is reduced to a single value.

Figure 5.7. Using reduce to sum the numbers in a stream



Let's take an in-depth look into how the reduce operation happens to sum a stream of numbers. First, 0 is used as the first parameter of the lambda (a), and 4 is consumed from the stream and used as the second parameter (b). 0 + 4 produces 4, and it becomes the new accumulated value. Then the lambda is called again with the accumulated value and the next element of the stream, 5, which produces the new accumulated value, 9. Moving forward, the lambda is called again with the accumulated value and the next element, 3, which produces 12. Finally, the lambda is called with 12 and the last element of the stream, 9, which produces the final value, 21.

You can make this code more concise by using a method reference. In Java 8 the Integer class now comes with a static sum method to add two numbers, which is just what you want instead of repeatedly writing out the same code as lambda:

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

No initial value

There's also an overloaded variant of reduce that doesn't take an initial value, but it returns an Optional object:

```
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

Why does it return an Optional<Integer>? Consider the case when the stream contains no elements. The reduce operation can't return a sum because it doesn't have an initial value. This is why the result is wrapped in an Optional object to indicate that the sum may be absent. Now see what else you can do with reduce.

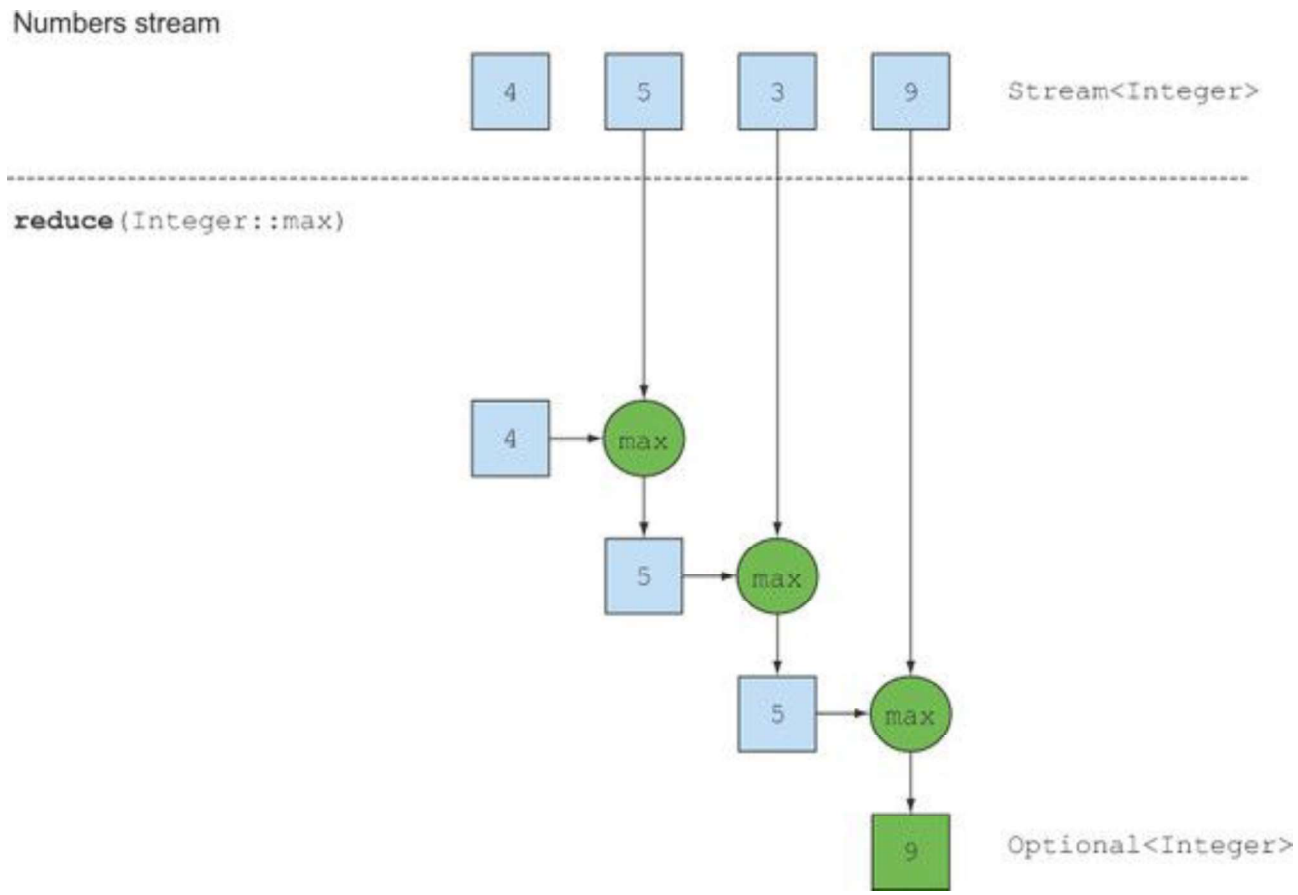
5.4.2. Maximum and minimum

It turns out that reduction is all you need to compute maxima and minima as well! Let's see how you can apply what you just learned about reduce to calculate the maximum or minimum element in a stream. As you saw, reduce takes two parameters:

- An initial value
- A lambda to combine two stream elements and produce a new value

The lambda is applied step by step to each element of the stream with the addition operator, as shown in figure 5.7. So you need a lambda that, given two elements, returns the maximum of them. The reduce operation will use the new value with the next element of the stream to produce a new maximum until the whole stream is consumed! You can use reduce as follows to calculate the maximum in a stream; this is illustrated in figure 5.8:

Figure 5.8. A reduce operation—calculating the maximum



Optional<Integer> max = numbers.stream().reduce(Integer::max);

To calculate the minimum, you need to pass Integer.min to the reduce operation instead of Integer.max:

Optional<Integer> min = numbers.stream().reduce(Integer::min);

You could have equally well used the lambda (x,y)->x<y?x:y instead of Integer::min, but the latter is easier to read.

To test your understanding of the reduce operation, have a go at Quiz 5.3.

Quiz 5.3: Reducing

How would you count the number of dishes in a stream using the map and reduce methods?

Answer:

You can solve this problem by mapping each element of a stream into the number 1 and then summing them using reduce! This is equivalent to counting in order the number of elements in the stream.

```
int count = menu.stream()
                    .map(d -> 1)
                    .reduce(0, (a, b) -> a + b);
```

A chain of map and reduce is commonly known as the map-reduce pattern, made famous by Google's use of it for web searching because it can be easily parallelized. Note that in chapter 4 you saw the built-in method count to count the number of elements in the stream:

```
long count = menu.stream().count();
```

Benefit of the reduce method and parallelism

The benefit of using reduce compared to the step-by-step iteration summation that you wrote earlier is that the iteration is abstracted using internal iteration, which enables the internal implementation to choose to perform the reduce operation in parallel. The iterative summation example involves shared updates to a sum variable, which doesn't parallelize gracefully. If you add in the needed synchronization, you'll likely discover that thread contention robs you of all the performance that parallelism was supposed to give you! Parallelizing this computation requires a different approach: partition the input, sum the partitions, and combine the sums. But now the code is starting to look really different. You'll see what this looks like in chapter 7 using the fork/join framework. But for now it's important to realize that the mutable accumulator pattern is a dead end for parallelization. You need a new pattern, and this is what reduce provides you. You'll also see in chapter 7 that to sum all the elements in parallel using streams, there's almost no modification to your code: stream() becomes parallelStream():

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

But there's a price to pay to execute this code in parallel, as we explain later: the lambda passed to reduce can't change state (for example, instance variables), and the operation needs to be associative so it can be executed in any order.

So far you saw reduction examples that produced an Integer: the sum of a stream, the maximum of a stream, or the number of elements in a stream. You'll see in section 5.6 that built-in methods such as sum and max are available as well to help you write slightly more concise code for common reduction patterns. We investigate a more complex form of reductions using the collect method in the next chapter. For example, instead of reducing a stream into an Integer, you can also reduce it into a Map if you want to group dishes by types.

Stream operations: stateless vs. stateful

You've seen a lot of stream operations. An initial presentation can make them seem a panacea; everything just works, and you get parallelism for free when you use parallelStream instead of stream to get a stream from a collection.

Certainly for many applications this is the case, as you've seen in the previous examples. You can turn a list of dishes into a stream, filter to select various dishes of a certain type, then map down the resulting stream to add on the number of calories, and then reduce to produce the total number of calories of the menu. You can even do such stream calculations in parallel. But these operations have different characteristics. There are issues about what internal state they need to operate.

Operations like map and filter take *each* element from the input stream and produce *zero or one* result in the output stream. These operations are thus in general *stateless*: they don't have an internal state (assuming the user-supplied lambda or method reference has no internal mutable state).

But operations like reduce, sum, and max need to have internal state to accumulate the result. In this case the internal state is small. In our example it consisted of an int or double. The internal state is of *bounded size* no matter how many elements are in the stream being processed.

By contrast, some operations such as sorted or distinct seem at first to behave like filter or map—all take a stream and produce another stream (an intermediate operation), but there's a crucial difference. Both sorting and removing duplicates from a stream require knowing the previous history to do their job. For example, sorting requires *all the elements to be buffered* before a single item can be added to the output stream; the storage requirement of the operation is *unbounded*. This can be problematic if the data stream is large or infinite. (What should reversing the stream of all prime numbers do? It should return the largest prime number, which mathematics tells us doesn't exist.) We call these operations *stateful operations*.

You’ve now seen a lot of stream operations that you can use to express sophisticated data processing queries! Table 5.1 summarizes the operations seen so far. You get to practice them in the next section through an exercise.

Table 5.1. Intermediate and terminal operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
distinct	Intermediate (stateful-unbounded)	Stream<T>		
skip	Intermediate (stateful-bounded)	Stream<T>	long	
limit	Intermediate (stateful-bounded)	Stream<T>	long	
map	Intermediate	Stream<R>	Function<T, R>	T -> R
flatMap	Intermediate	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	Intermediate (stateful-unbounded)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	Terminal	boolean	Predicate<T>	T -> boolean
noneMatch	Terminal	boolean	Predicate<T>	T -> boolean
allMatch	Terminal	boolean	Predicate<T>	T -> boolean
findAny	Terminal	Optional<T>		
findFirst	Terminal	Optional<T>		

Operation	Type	Return type	Type/functional interface used	Function descriptor
forEach	Terminal	void	Consumer<T>	T -> void
collect	terminal	R	Collector<T, A, R>	
reduce	Terminal (stateful-bounded)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	Terminal	long		

5.5. Putting it all into practice

In this section, you get to practice what you've learned about streams so far. We give a different domain: traders executing transactions. You're asked by your manager to find answers to eight queries. Can you do it? We give the solutions in section 5.5.2, but you should try them yourself first to get some practice.

1. Find all transactions in the year 2011 and sort them by value (small to high).
2. What are all the unique cities where the traders work?
3. Find all traders from Cambridge and sort them by name.
4. Return a string of all traders' names sorted alphabetically.
5. Are any traders based in Milan?
6. Print all transactions' values from the traders living in Cambridge.
7. What's the highest value of all the transactions?
8. Find the transaction with the smallest value.

5.5.1. The domain: Traders and Transactions

Here's the domain you'll be working with, a list of Traders and Transactions:

```
Trader raoul = new Trader("Raoul", "Cambridge");
```

```
Trader mario = new Trader("Mario","Milan");
Trader alan = new Trader("Alan","Cambridge");
Trader brian = new Trader("Brian","Cambridge");

List<Transaction> transactions = Arrays.asList(
    new Transaction(brian, 2011, 300),
    new Transaction(raoul, 2012, 1000),
    new Transaction(raoul, 2011, 400),
    new Transaction(mario, 2012, 710),
    new Transaction(mario, 2012, 700),
    new Transaction(alan, 2012, 950)
);
```

Trader and Transaction are classes defined as follows:

```
public class Trader{

    private final String name;
    private final String city;

    public Trader(String n, String c){
        this.name = n;
        this.city = c;
    }

    public String getName(){
        return this.name;
    }

    public String getCity(){
        return this.city;
    }

    public String toString(){
        return "Trader:"+this.name + " in " + this.city;
    }
}

public class Transaction{
```

```
    private final Trader trader;
    private final int year;
    private final int value;

    public Transaction(Trader trader, int year, int value){
        this.trader = trader;
        this.year = year;
        this.value = value;
    }

    public Trader getTrader(){
        return this.trader;
    }

    public int getYear(){
        return this.year;
    }

    public int getValue(){
        return this.value;
    }

    public String toString(){
        return "{" + this.trader + ", " +
                "year: "+this.year+", " +
                "value:" + this.value +"}";
    }
}
```

5.5.2. Solutions

We now provide the solutions in the following code listings, so you can verify your understanding of what you've learned so far. Well done!

Listing 5.1. Find all transactions in 2011 and sort by value (small to high)

```
List<Transaction> tr2011 =
    transactions.stream()
                .filter(transaction -> transaction.getYear() == 2011)
                .sorted(comparing(Transaction::getValue))
                .collect(toList());
```

Pass a predicate to `filter` to select transactions in year 2011.

Collect all the elements of the resulting `Stream` into a `List`.

Sort them by using the value of the transaction.

Listing 5.2. What are all the unique cities where the traders work?

```
List<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .distinct()
                .collect(toList());
```

Extract the city from each trader associated with the transaction.

Select only unique cities.

You haven't seen this yet, but you could also drop distinct() and use toSet() instead, which would convert the stream into a set. You'll learn more about it in chapter 6.

```
Set<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .collect(toSet());
```

Listing 5.3. Find all traders from Cambridge and sort them by name

```
List<Trader> traders =
    transactions.stream()
                .map(Transaction::getTrader)
                .filter(trader -> trader.getCity().equals("Cambridge"))
                .distinct()
                .sorted(comparing(Trader::getName))
                .collect(toList());
```

Extract all traders from the transactions.

Select only the traders from Cambridge.

Make sure you don't have any duplicates.

Sort the resulting stream of traders by their names.

Listing 5.4. Return a string of all traders' names sorted alphabetically

```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .reduce("", (n1, n2) -> n1 + n2);
```

Extract all the names of the traders as a `Stream` of `Strings`.

Select only the unique names.

Sort the names alphabetically.

Combine each name one by one to form a `String` that concatenates all the names.

Note that this solution isn't very efficient (all Strings are repeatedly concatenated, which creates a new String object at each iteration). In the next chapter, you'll see a more efficient solution that uses joining() as follows (which internally makes use of a StringBuilder):

```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .collect(joining());
```

Listing 5.5. Are any traders based in Milan?

```
boolean milanBased =
    transactions.stream()
                .anyMatch(transaction -> transaction.getTrader()
                                         .getCity()
                                         .equals("Milan"));
```

Pass a predicate to `anyMatch` to check if there's a trader from Milan.

Listing 5.6. Print all transactions' values from the traders living in Cambridge

```
transactions.stream()
    .filter(t -> "Cambridge".equals(t.getTrader().getCity()))
    .map(Transaction::getValue)
    .forEach(System.out::println);
```

Select the transactions where the traders live in Cambridge.

Extract the values of these trades.

Print each value.

Listing 5.7. What's the highest value of all the transactions?

```
Optional<Integer> highestValue =
    transactions.stream()
                .map(Transaction::getValue)
                .reduce(Integer::max);
```

Extract the value of each transaction.

Calculate the max of the resulting stream.

Listing 5.8. Find the transaction with the smallest value

```
Optional<Transaction> smallestTransaction =
  transactions.stream()
              .reduce((t1, t2) ->
                        t1.getValue() < t2.getValue() ? t1 : t2);
```

Find the smallest transaction by repeatedly comparing the values of each transaction.

You can do better. A stream supports the methods min and max that take a Comparator as argument to specify which key to compare with when calculating the minimum or maximum:

```
Optional<Transaction> smallestTransaction =
  transactions.stream()
                .min(comparing(Transaction::getValue));
```

5.6. Numeric streams

You saw earlier that you could use the reduce method to calculate the sum of the elements of a stream. For example, you can calculate the number of calories in the menu as follows:

```
int calories = menu.stream()
                    .map(Dish::getCalories)
                    .reduce(0, Integer::sum);
```

The problem with this code is that there's an insidious boxing cost. Behind the scenes each Integer needs to be unboxed to a primitive before performing the summation. In addition, wouldn't it be nicer if you could call a sum method directly as follows?

```
int calories = menu.stream()
                    .map(Dish::getCalories)
                    .sum();
```

But this isn't possible. The problem is that the method map generates a Stream<T>. Even though the elements of the stream are of type Integer, the Streams interface doesn't define a sum method. Why not? Say you had only a Stream<Dish> like the menu; it wouldn't make any sense to be able to sum dishes. But don't worry; the Streams API also supplies *primitive stream specializations* that support specialized methods to work with streams of numbers.

5.6.1. Primitive stream specializations

Java 8 introduces three primitive specialized stream interfaces to tackle this issue, IntStream, DoubleStream, and LongStream, that respectively specialize the elements of a stream to be int, long, and double—and thereby avoid hidden boxing costs. Each of these interfaces brings new methods to perform common numeric reductions such as sum to calculate the sum of a numeric stream and max to find the maximum element. In addition, they have methods to convert back to a stream of objects when necessary. The thing to remember is that these specializations aren't more complexity about streams but instead more complexity caused by boxing—the (efficiency-based) difference between int and Integer and so on.

Mapping to a numeric stream

The most common methods you'll use to convert a stream to a specialized version are mapToInt, mapToDouble, and mapToLong. These methods work exactly like the method map that you saw earlier but return a specialized stream instead of a Stream<T>. For example, you can use mapToInt as follows to calculate the sum of calories in the menu:

```
int calories = menu.stream()
                   .mapToInt(Dish::getCalories)
                   .sum();
```

Returns an IntStream

Returns a Stream<Dish>

Here, the method mapToInt extracts all the calories from each dish (represented as an Integer) and returns an IntStream as the result (rather than a Stream<Integer>). You can then call the sum method defined on the IntStream interface to calculate the sum of calories! Note that if the stream were empty, sum would return 0 by default. IntStream also supports other convenience methods such as max, min, and average.

Converting back to a stream of objects

Similarly, once you have a numeric stream, you may be interested in converting it back to a nonspecialized stream. For example, the operations of an IntStream are restricted to produce primitive integers: the map operation of an IntStream takes a lambda that takes an int and produces an int (an IntUnaryOperator). But you may want to produce a different value such as a Dish. For this you need to access the operations defined in the Streams interface that are more general. To convert from a primitive stream to a general stream (each int will be boxed to an Integer) you can use the method boxed as follows:

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> stream = intStream.boxed();
```

Converting a Stream to a numeric stream

Converting the numeric stream to a Stream

You'll learn in the next section that boxed is particularly useful when you deal with numeric ranges that need to be boxed into a general stream.

Default values: OptionalInt

The sum example was convenient because it has a default value: 0. But if you want to calculate the maximum element in an IntStream, you need something different because 0 is a wrong result. How can you differentiate that the stream has no element and that the real maximum is 0? Earlier we introduced the Optional class, which is a container that indicates the presence or absence of a value. Optional can be parameterized with reference types such as Integer, String, and so on. There's a primitive specialized version of Optional as well for the three primitive stream specializations: OptionalInt, OptionalDouble, and OptionalLong.

For example, you can find the maximal element of an IntStream by calling the max method, which returns an OptionalInt:

```
OptionalInt maxCalories = menu.stream()
                              .mapToInt(Dish::getCalories)
                              .max();
```

You can now process the OptionalInt explicitly to define a default value if there's no maximum:

```
int max = maxCalories.orElse(1);
```

Provide an explicit default maximum if there's no value.

5.6.2. Numeric ranges

A common use case when dealing with numbers is working with ranges of numeric values. For example, suppose you'd like to generate all numbers between 1 and 100. Java 8 introduces two static methods available on IntStream and LongStream to help generate such ranges: range and rangeClosed. Both methods take the starting value of the range as the first parameter and the end value of the range as the second parameter. But range is exclusive, whereas rangeClosed is inclusive. Let's look at an example:

```
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
                                .filter(n -> n % 2 == 0);

System.out.println(evenNumbers.count());
```

Represents the range [1, 100].

A stream of even numbers from 1 to 100.

There are 50 even numbers from 1 to 100.

Here you use the rangeClosed method to generate a range of all numbers from 1 to 100. It produces a stream so you can chain the filter method to select only even numbers. At this stage no computation has been done. Finally, you call count on the resulting stream. Because count is a terminal operation, it will process the stream and return the result 50, which is the number of even numbers from 1 to 100, inclusive. Note that by comparison, if you were using IntStream.range(1, 100) instead, the result would be 49 even numbers because range is exclusive.

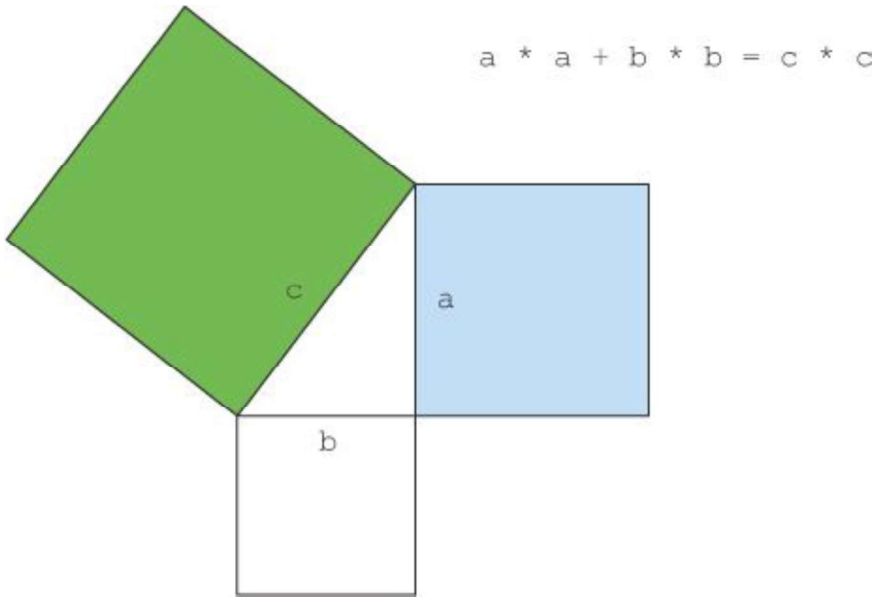
5.6.3. Putting numerical streams into practice: Pythagorean triples

We now look at a more difficult example so you can solidify what you've learned about numeric streams and all the stream operations you've learned so far. Your mission, if you choose to accept it, is to create a stream of Pythagorean triples.

Pythagorean triple

So what's a Pythagorean triple? We have to go back a few years in the past. In one of your exciting math classes, you learned that the famous Greek mathematician Pythagoras discovered that certain triples of numbers (a, b, c) satisfy the formula a * a + b * b = c * c where a, b, and c are integers. For example, (3, 4, 5) is a valid Pythagorean triple because 3 * 3 + 4 * 4 = 5 * 5 or 9 + 16 = 25. There are an infinite number of such triples. For example, (5, 12, 13), (6, 8, 10), and (7, 24, 25) are all valid Pythagorean triples. Such triples are useful because they describe the three side lengths of a right-angled triangle, as illustrated in figure 5.9.

Figure 5.9. The Pythagorean theorem



Representing a triple

So where do you start? The first step is to define a triple. Instead of (more properly) defining a new class to represent a triple, you can use an array of int with three elements, for example, new int[]{3, 4, 5} to represent the tuple (3, 4, 5). You can now access each individual component of the tuple using array indexing.

Filtering good combinations

Let's assume someone provides you with the first two numbers of the triple: a and b. How do you know whether that will form a good combination? You need to test whether the square root of a * a + b * b is an integer number; that is, it has no fractional part, which in Java can be expressed using expr % 1.0. If it's not an integer, that means c is not an integer. You can express this requirement as a filter operation (you'll see how to connect it later to form valid code):

filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)

Assuming that surrounding code has given a value for a and assuming stream provides possible values for b, filter will select only those values for b that can form a Pythagorean triple with a. You may be wondering what the line Math.sqrt(a*a + b*b) % 1 == 0 is about. It's basically a way to test whether Math.sqrt(a*a + b*b) returns an integer result. The condition will fail if the result of the square root produces a number with a decimal such as 9.1 (9.0 is valid).

Generating tuples

Following the filter, you know that both a and b can form a correct combination. You now need to create a triple. You can use the map operation to transform each element into a Pythagorean triple as follows:

```
stream.filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
        .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Generating b values

You're getting closer! You now need to generate values for b. You saw that Stream .rangeClosed allows you to generate a stream of numbers in a given interval. You can use it to provide numeric values for b, here 1 to 100:

```
IntStream.rangeClosed(1, 100)
        .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
        .boxed()
        .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Note that you call boxed after the filter to generate a Stream<Integer> from the IntStream returned by rangeClosed. This is because your map returns an array of int for each element of the stream. The map method from an IntStream expects only another int to be returned for each element of the stream, which isn't what you want! You can rewrite this using the method mapToObj of an IntStream, which returns an object-valued stream:

```
IntStream.rangeClosed(1, 100)
        .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
        .mapToObj(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Generating a values

There's one crucial piece that we assumed was given: the value for a. You now have a stream that produces Pythagorean triples provided the value a is known. How can you fix this? Just like with b, you need to generate numeric values for a! The final solution is as follows:

```
Stream<int[]> pythagoreanTriples =
    IntStream.rangeClosed(1, 100).boxed()
            .flatMap(a ->
```

```
IntStream.rangeClosed(a, 100)
        .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
        .mapToObj(b ->
            new int[]{a, b, (int)Math.sqrt(a * a + b * b)})
    );
```

Okay, what's the flatMap about? First, you create a numeric range from 1 to 100 to generate values for a. For each given value of a you're creating a stream of triples. Mapping a value of a to a stream of triples would result in a stream of streams! The flatMap method does the mapping and also flattens all the generated streams of triples into a single stream. As a result you produce a stream of triples. Note also that you change the range of b to be a to 100. There's no need to start the range at the value 1 because this would create duplicate triples (for example, (3, 4, 5) and (4, 3, 5)).

Running the code

You can now run your solution and select explicitly how many triples you'd like to return from the generated stream using the limit operation that you saw earlier:

```
pythagoreanTriples.limit(5)
                .forEach(t ->
                    System.out.println(t[0] + ", " + t[1] + ", " + t[2]));
```

This will print

```
3, 4, 5
5, 12, 13
6, 8, 10
7, 24, 25
8, 15, 17
```

Can you do better?

The current solution isn't optimal because you calculate the square root twice. One possible way to make your code more compact is to generate all triples of the form (a*a, b*b, a*a+b*b) and then filter the ones that match your criteria:

```
Stream<double[]> pythagoreanTriples2 =
  IntStream.rangeClosed(1, 100).boxed()
           .flatMap(a ->
              IntStream.rangeClosed(a, 100)
                       .mapToObj(
                          b -> new double[]{a, b, Math.sqrt(a*a + b*b)})
                       .filter(t -> t[2] % 1 == 0));
```

Produce triples.

The third element of the tuple must be an integer.

5.7. Building streams

Hopefully by now you're convinced that streams are very powerful and useful to express data processing queries. So far, you were able to get a stream from a collection using the stream method. In addition, we showed you how to create numerical streams from a range of numbers. But you can create streams in many more ways! This section shows how you can create a stream from a sequence of values, from an array, from a file, and even from a generative function to create infinite streams!

5.7.1. Streams from values

You can create a stream with explicit values by using the static method Stream.of, which can take any number of parameters. For example, in the following code you create a stream of strings directly using Stream.of. You then convert the strings to uppercase before printing them one by one:

Stream<String> stream = ***Stream.of("Java 8 ", "Lambdas ", "In ", "Action")***;
stream.map(String::toUpperCase).forEach(System.out::println);

You can get an empty stream using the empty method as follows:

Stream<String> emptyStream = ***Stream.empty()***;

5.7.2. Streams from arrays

You can create a stream from an array using the static method Arrays.stream, which takes an array as parameter. For example, you can convert an array of primitive ints into an IntStream as follows:

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```

The sum is 41.

5.7.3. Streams from files

Java's NIO API (non-blocking I/O), which is used for I/O operations such as processing a file, has been updated to take advantage of the Streams API. Many static methods in java.nio.file.Files return a stream. For example, a useful method is Files.lines, which returns a stream of lines as strings from a given file. Using what you've learned so far, you could use this method to find out the number of unique words in a file as follows:

```
long uniqueWords = 0;
try(Stream<String> lines =
          Files.lines(Paths.get("data.txt"), Charset.defaultCharset())){
uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))
                   .distinct()
                   .count();
}
catch(IOException e){

}
```

Streams are autoclosable.

Generate a stream of words.

Remove duplicates.

Count the number of unique words.

Deal with the exception if one occurs when opening the file.

You use Files.lines to return a stream where each element is a line in the given file. You then split each line into words by calling the split method on line. Notice how you use flatMap to produce one flattened stream of words instead of multiple streams of words for each line. Finally, you count each distinct word in the stream by chaining the methods distinct and count.

5.7.4. Streams from functions: creating infinite streams!

The Streams API provides two static methods to generate a stream from a function: Stream.iterate and Stream.generate. These two operations let you create what we call an *infinite stream*: a stream that doesn't have a fixed size like when you create a stream from a fixed collection. Streams produced by iterate and generate create values on demand given a function and can therefore calculate values forever! It's generally sensible to use limit(n) on such streams to avoid printing an infinite number of values.

Iterate

Let's look at a simple example of how to use iterate before we explain it:

```
Stream.iterate(0, n -> n + 2)
        .limit(10)
        .forEach(System.out::println);
```

The iterate method takes an initial value, here 0, and a lambda (of type Unary-Operator<T>) to apply successively on each new value produced. Here you return the previous element added with 2 using the lambda n -> n + 2. As a result, the iterate method produces a stream of all even numbers: the first element of the stream is the initial value 0. Then it adds 2 to produce the new value 2; it adds 2 again to produce the new value 4 and so on. This iterate operation is fundamentally sequential because the result depends on the previous application. Note that this operation produces an *infinite stream*—the stream doesn't have an end because values are computed on demand and can be computed forever. We say the stream is *unbounded*. As we discussed earlier, this is a key difference between a stream and a collection. You're using the limit method to explicitly limit the size of the stream. Here you select only the first 10 even numbers. You then call the forEach terminal operation to consume the stream and print each element individually.

In general, you should use iterate when you need to produce a sequence of successive values, for example, a date followed by its next date: January 31, February 1, and so on. To see a more difficult example of how you can apply iterate, try out Quiz 5.4.

Quiz 5.4: Fibonacci tuples series

The Fibonacci series is famous as a classic programming exercise. The numbers in the following sequence are part of the Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.... The first two numbers of the series are 0 and 1, and each subsequent number is the sum of the previous two.

The series of Fibonacci tuples is similar; you have a sequence of a number and its successor in the series: (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21)....

Your task is to generate the first 20 elements of the series of Fibonacci tuples using the iterate method!

Let us help you get started. The first problem is that the iterate method takes a UnaryOperator<T> as argument and you need a stream of tuples such as (0, 1). You can, again rather sloppily, use an array of two elements to represent a tuple. For example, new int[]{0, 1} represents the first element of the Fibonacci series (0, 1). This will be the initial value of the iterate method:

```
Stream.iterate(new int[]{0, 1}, ???)
        .limit(20)
        .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));
```

In this quiz, you need to figure out the highlighted code with the ???. Remember that iterate will apply the given lambda successively.

Answer:

```
Stream.iterate(new int[]{0, 1},
                    t -> new int[]{t[1], t[0]+t[1]})
        .limit(20)
        .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));
```

How does it work? iterate needs a lambda to specify the successor element. In the case of the tuple (3, 5) the successor is (5, 3+5) = (5, 8). The next one is (8, 5+8). Can you see the pattern? Given a tuple, the successor is (t[1], t[0] + t[1]). This is what the following lambda specifies: t -> new int[]{t[1],t[0] + t[1]}. By running this code you'll get the series (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21).... Note that if you just wanted to print the normal Fibonacci series, you could use a map to extract only the first element of each tuple:

```
Stream.iterate(new int[]{0, 1},
                    t -> new int[]{t[1],t[0] + t[1]})
        .limit(10)
        .map(t -> t[0])
        .forEach(System.out::println);
```

This code will produce the Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34....

Generate

Similarly to the method iterate, the method generate lets you produce an infinite stream of values computed on demand. But generate doesn't apply successively a function on each new produced value. It takes a lambda of type Supplier<T> to provide new values. Let's look at an example of how to use it:

```
Stream.generate(Math::random)
        .limit(5)
        .forEach(System.out::println);
```

This code will generate a stream of five random double numbers from 0 to 1. For example, one run gives the following:

0.9410810294106129
0.6586270755634592
0.9592859117266873
0.13743396659487006
0.3942776037651241

The static method Math.random is used as a generator for new values. Again you limit the size of the stream explicitly using the limit method; otherwise the stream would be unbounded!

You may be wondering if there's anything else useful you can do using the method generate. The supplier we used (a method reference to Math.random) was stateless: it wasn't recording any values somewhere that can be used in later computations. But a supplier doesn't have to be stateless. You can create a supplier that stores state that it can modify and use when generating the next value of the stream. As an example, we show how you can also create the Fibonacci series from Quiz 5.4 using generate so you can compare it with the approach using the iterate method! But it's important to note that a supplier that's stateful isn't safe to use in parallel code. So what follows is shown just for completeness but should be avoided! We discuss the problem of operations with side effects and parallel streams further in chapter 7.

We'll use an IntStream in our example to illustrate code that's designed to avoid boxing operations. The generate method on IntStream takes an IntSupplier instead of a Supplier<T>. For example, here's how to generate an infinite stream of ones:

```
IntStream ones = IntStream.generate(() -> 1);
```

You saw in the chapter 3 that lambdas let you create an instance of a functional interface by providing the implementation of the method directly inline. You can also pass an explicit object as follows by implementing the getAsInt method defined in the IntSupplier interface (although this seems gratuitously long-winded, please bear with us):

```
IntStream twos = IntStream.generate(new IntSupplier(){
            public int getAsInt(){
                return 2;
            }
        });
```

The generate method will use the given supplier and repeatedly call the getAsInt method, which always returns 2. But the difference between the anonymous class used here and a lambda is that the anonymous class can define state via fields, which the getAsInt method can modify. This

is an example of a side effect. All lambdas you've seen so far were side-effect free; they didn't change any state.

To come back to our Fibonacci tasks, what you need to do now is create an IntSupplier that maintains in its state the previous value in the series, so getAsInt can use it to calculate the next element. In addition, it can update the state of the IntSupplier for the next time it's called. The following code shows how to create an IntSupplier that will return the next Fibonacci element when it's called:

```
IntSupplier fib = new IntSupplier(){
    private int previous = 0;
    private int current = 1;
    public int getAsInt(){
        int oldPrevious = this.previous;
        int nextValue = this.previous + this.current;
        this.previous = this.current;
        this.current = nextValue;
        return oldPrevious;
    }
};
IntStream.generate(fib).limit(10).forEach(System.out::println);
```

In the preceding code you create an instance of IntSupplier. This object has *mutable* state: it tracks the previous Fibonacci element and the current Fibonacci element in two instance variables. The getAsInt method changes the state of the object when it's called so that it produces new values on each call. In comparison, our approach using iterate was purely *immutable*: you didn't modify existing state but were creating new tuples at each iteration. You'll learn in chapter 7 that you should always prefer an *immutable approach* in order to process a stream in parallel and expect a correct result.

Note that because you're dealing with a stream of infinite size, you have to limit its size explicitly using the operation limit; otherwise, the terminal operation (in this case forEach) will compute forever. Similarly, you can't sort or reduce an infinite stream because all elements need to be processed, but this would take forever because the stream is infinite!

5.8. Summary

It's been a long but rewarding chapter! You can now process collections more effectively. Indeed, streams let you express sophisticated data processing queries concisely. In addition, streams can be parallelized transparently. Here are some key concepts to take away from this chapter:

- The Streams API lets you express complex data processing queries. Common stream operations are summarized in table 5.1.
- You can filter and slice a stream using the filter, distinct, skip, and limit methods.
- You can extract or transform elements of a stream using the map and flatMap methods.
- You can find elements in a stream using the findFirst and findAny methods. You can match a given predicate in a stream using the allMatch, noneMatch, and anyMatch methods.
- These methods make use of short-circuiting: a computation stops as soon as a result is found; there's no need to process the whole stream.
- You can combine all elements of a stream iteratively to produce a result using the reduce method, for example, to calculate the sum or find the maximum of a stream.
- Some operations such as filter and map are stateless; they don't store any state. Some operations such as reduce store state to calculate a value. Some operations such as sorted and distinct also store state because they need to buffer all the elements of a stream before returning a new stream. Such operations are called *stateful operations*.
- There are three primitive specializations of streams: IntStream, DoubleStream, and LongStream. Their operations are also specialized accordingly.
- Streams can be created not only from a collection but also from values, arrays, files, and specific methods such as iterate and generate.
- An infinite stream is a stream that has no fixed size.