Part 2

Functional-style data processing with streams

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.

4

Introducing streams

This chapter covers

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

What would you do without collections in Java? 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 are tasked to create a collection of dishes to represent a menu to calculate different queries. For example, you may want to find out the total number of calories for the menu. Or, you may need to simply filter the menu 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 (or `filter') the names of dishes that are low in calories: `SELECT name FROM dishes WHERE calorie < 400`. As you can see, in SQL you don't need to implement *how* to filter using the `calorie` attribute of a dish (as you would with Java collections, e.g. using an iterator and an accumulator). Instead, you just write *what* you want as result. 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 dish: menu) {
    if(dish.getCalories() < 400) {                              ❶
        lowCaloricDishes.add(dish);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {     ❷
    public int compare(Dish dish1, Dish dish2) {
        return Integer.compare(dish1.getCalories(), dish2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish dish: lowCaloricDishes) {
    lowCaloricDishesName.add(dish.getName());                   ❸
}
```

❶ **Filter the elements using an accumulator.**
❷ **Sort the dishes with an anonymous class.**
❸ **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)        (1)
    .sorted(comparing(Dish::getCalories))      (2)
    .map(Dish::getName)                        (3)
    .collect(toList());                        (4)
```

(1) Select dishes that are below 400 calories.
(2) Sort them by calories.
(3) Extract the names of these dishes.
(4) 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? Should you actually use this method at all? 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. Another way to think about the benefit of this approach is that the threading model is decoupled from the query itself. Because you are providing a recipe for a query, it could be executed sequentially or in parallel. You will learn more about this in Chapter 7.
- 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.

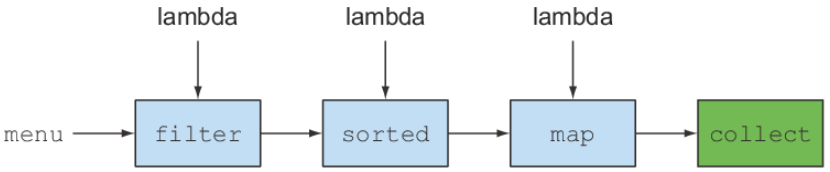


Figure 4.1 Chaining stream operations forming a stream pipeline

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. But don't waste too much of your time – instead 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()                                          (1)
      .filter(dish -> dish.getCalories() > 300)          (2)
      .map(Dish::getName)                                (3)
      .limit(3)                                          (4)
      .collect(toList());                                (5)
System.out.println(threeHighCaloricDishNames);           (6)
```

(1) **Get a stream from menu (the list of dishes).**
(2) **Create a pipeline of operations: first filter high-calorie dishes.**
(3) **Get the names of the dishes.**
(4) **Select only the first three.**
(5) **Store the results in another List.**
(6) **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:

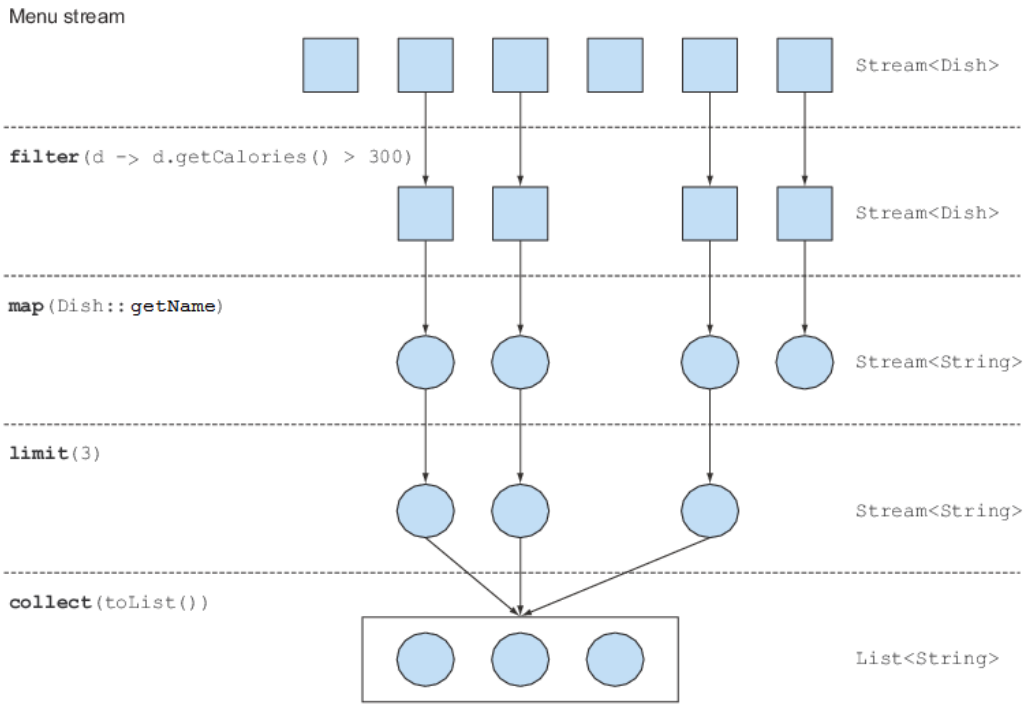


Figure 4.2 Filtering a menu using a stream to find out three high-calorie dish names

- `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.

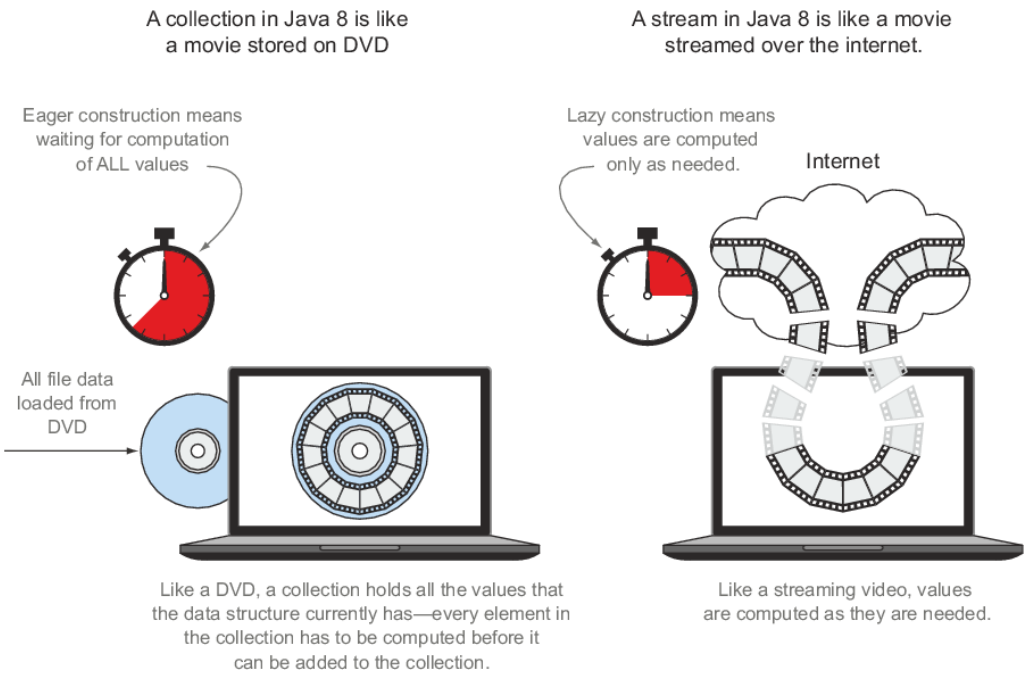


Figure 4.3 Streams vs. collections

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("Modern", "Java", "In", "Action");
Stream<String> s = title.stream();
s.forEach(System.out::println);                          (1)
s.forEach(System.out::println);                          (2)
```

(1) Prints each word in the title.
(2) 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 dish: menu) {                                   (1)
    names.add(dish.getName());                           (2)
}
```

(1) Explicitly iterate the list of menu sequentially.
(2) 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()) {                              (1)
    Dish dish = iterator.next();
```

```
    names.add(dish.getName());
}
```

❶ Iterating explicitly

Listing 4.3 Streams: internal iteration

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

❶ Parameterize map with the getName method to extract the name of a dish.
❷ Start executing the pipeline of operations; no iteration!

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).

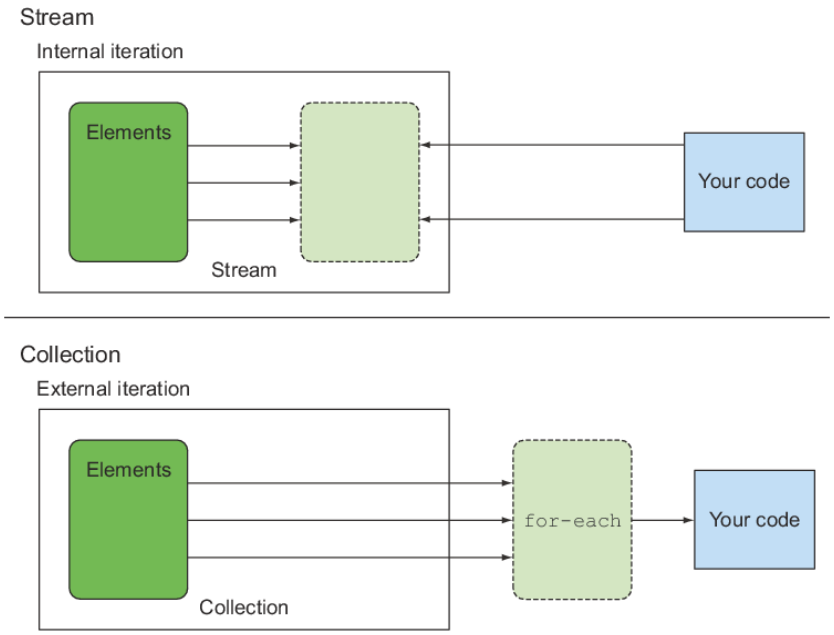


Figure 4.4 Internal vs. external iteration

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.

Quiz 4.1: External vs internal iteration

Based on what you learnt about external iteration in Listing 4.1 and 4.2, which Stream operation would you use to refactor the following code?

```
List<String> highCaloricDishes = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish dish = iterator.next();
    if(dish.getCalories() > 300) {
        highCaloricDishes.add(d.getName());
    }
```

```
}
```

Answer:

You need to use the `filter` pattern

```
List<String> highCaloricDish =
    menu.stream()
        .filter(dish -> dish.getCalories() > 300)
        .collect(toList());
```

Don't worry if you are still unfamiliar with how to precisely write a stream query, you will learn this in more details 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()                                  (1)
                         .filter(dish -> dish.getCalories() > 300)  (2)
                         .map(Dish::getName)                        (3)
                         .limit(3)                                  (4)
                         .collect(toList());                        (5)
```

(1) **Get a stream from the list of dishes.**
(2) **Intermediate operation.**
(3) **Intermediate operation.**
(4) **Intermediate operation.**
(5) **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?

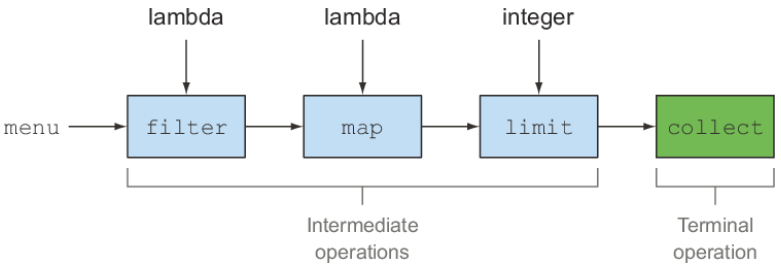


Figure 4.5 Intermediate vs. terminal operations

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(dish -> {
                        System.out.println("filtering:" + dish.getName());
                        return dish.getCalories() > 300;
                    })                                                    ❶
        .map(dish -> {
                        System.out.println("mapping:" + dish.getName());
                        return dish.getName();
                })                                                        ❷
        .limit(3)
        .collect(toList());
System.out.println(names);
```

❶ Printing the dishes as they're filtered
❷ 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]
```

By doing this you can notice that the `Stream` library performs several optimizations exploiting 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 (compiler experts 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 4.2: Intermediate vs. terminal operations

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

```
long count = menu.stream()
                .filter(dish -> dish.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.[16] 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).

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	Return type	Purpose
`forEach`	Terminal	`void`	Consumes each element from a stream and applies a lambda to each of them.
`count`	Terminal	`long`	Returns the number of elements in a stream.
`collect`	Terminal	(generic)	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.

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

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 (`lazily').

5

Working with streams

This chapter covers

- Filtering, slicing, and mapping
- 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. You will learn about operations available in Java 8 but also new additions in Java 9. 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

In this section, we look at how to select elements of a stream: filtering with a predicate, and filtering only unique elements.

5.1.1 Filtering with a predicate

The `Stream` 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:

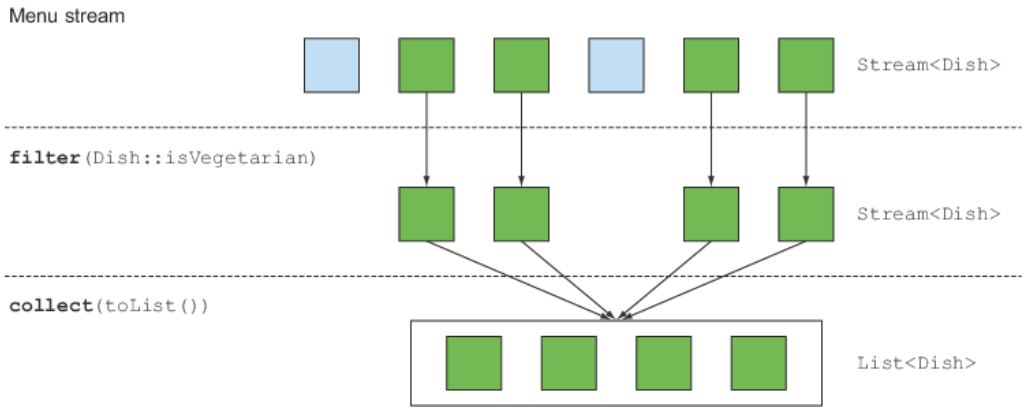


Figure 5.1 Filtering a stream with a predicate

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

❶ A method reference to check if a dish is vegetarian friendly

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 then eliminates duplicates (using the equals() method for comparison). Figure 5.2 shows this visually:

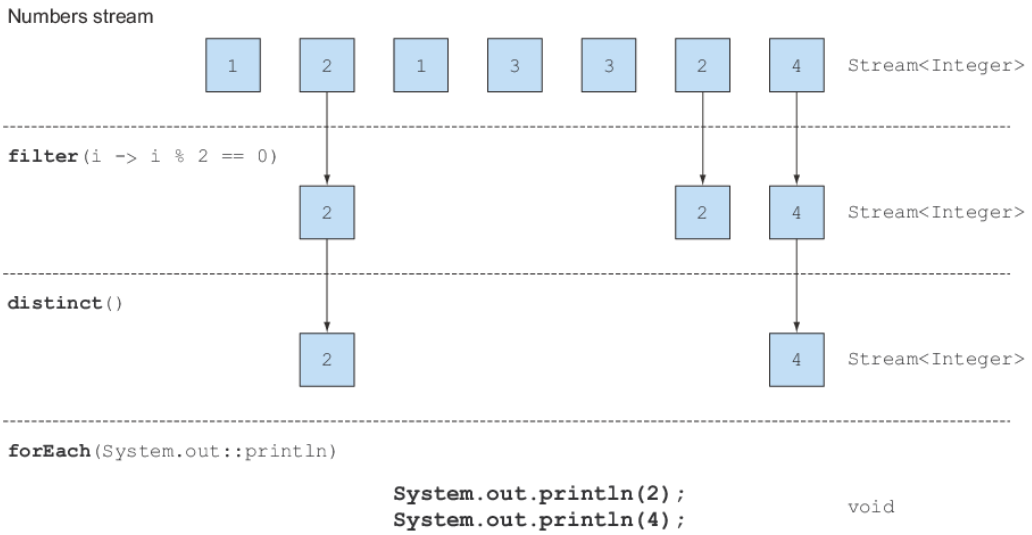


Figure 5.2 Filtering unique elements in a stream

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

5.2 Slicing a stream

In this section, you will learn how to select and skip elements in a stream in different ways. There are operations available that let you efficiently select or drop elements using a predicate, ignore the first few elements of a stream, or truncate a stream to a given size.

5.2.1 Slicing using a predicate

Java 9 added two new methods which are useful for efficiently selecting elements in a stream: `takeWhile` and `dropWhile`.

USING TAKEWHILE

Let's say you have the following special list of dishes:

```
List<Dish> specialMenu = Arrays.asList(
    new Dish("seasonal fruit", true, 120, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
    new Dish("french fries", true, 530, Dish.Type.OTHER));
```

How would you select the dishes that have fewer than 320 calories? Instinctively, you know already from the previous section that the operation `filter` can be used as follows:

```
List<Dish> filteredMenu
    = specialMenu.stream()
                 .filter(dish -> dish.getCalories() < 320)
                 .collect(toList()); ❶
```

❶ **: [seasonal fruit, prawns]**

However, you will notice that the initial list was already sorted on the number of calories! The downside of using the `filter` operation here is that you need to iterate through the whole stream and the predicate is applied to each element. Instead, you could just stop once you found a dish that is greater than (or equal to) 320 calories. With a small list this may not seem like a huge benefit but it can become handy useful if you work with potentially large stream of elements. But how do you specify this? The `takeWhile` operation is here to rescue you! It lets

you slice any stream (even an infinite stream as you will learn later) using a predicate. But thankfully it stops once it has found an element that fails to match. Here's how you can use it:

```
List<Dish> slicedMenu1
    = specialMenu.stream()
                 .takeWhile(dish -> dish.getCalories() < 320)
                 .collect(toList()); # A
```

❶ **[seasonal fruit, prawns]**

USING DROPWHILE

How about getting the other elements though? In other words, how about finding the elements that have greater than 320 calories? You can use the `dropWhile` operation for this:

```
List<Dish> slicedMenu2
    = specialMenu.stream()
                 .dropWhile(dish -> dish.getCalories() < 320)
                 .collect(toList()); # A
```

❶ **[rice, chicken, french fries]**

The `dropWhile` operation is the complement of `takeWhile`. It throws away the elements at the start where the predicate is false. Once the predicate evaluates to true it stops and returns all the remaining elements, and it even works if there are an infinite number of remaining elements!

5.2.2 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 = specialMenu
                        .stream()
                        .filter(dish -> dish.getCalories() > 300)
                        .limit(3)
                        .collect(toList()); ❶
```

❶ **[rice, chicken, french fries]**

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.

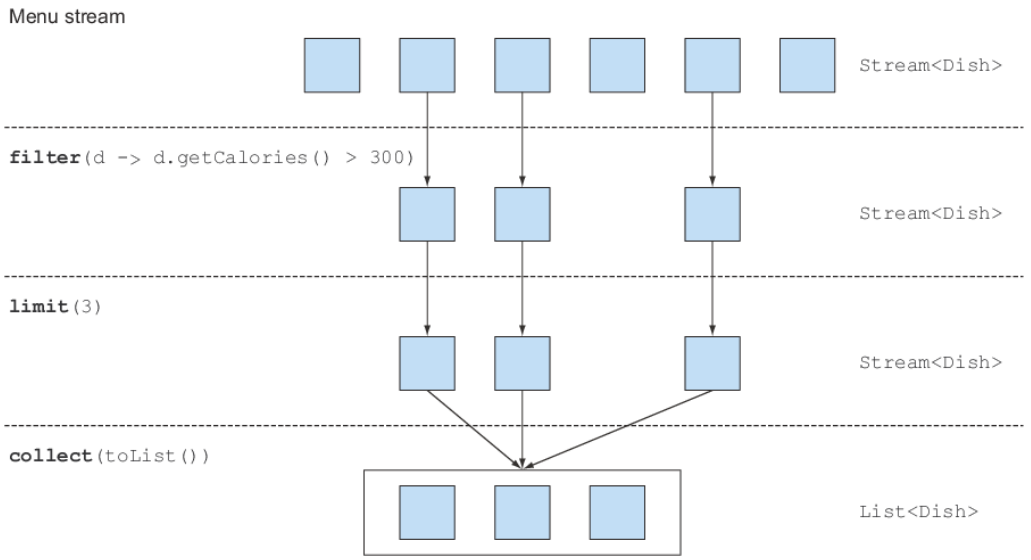


Figure 5.3 Truncating a stream

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`.

5.2.3 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:

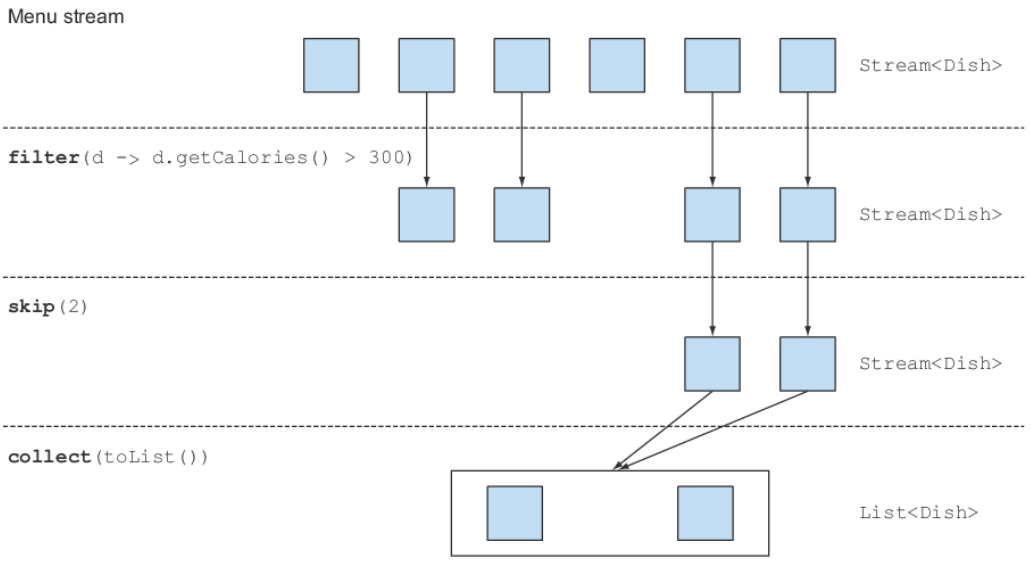


Figure 5.4 Skipping elements in a stream

```
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(dish -> dish.getType() == Dish.Type.MEAT)
        .limit(2)
        .collect(toList());
```

5.3 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.

5.3.1 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("Modern", "Java", "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());
```

5.3.2 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.

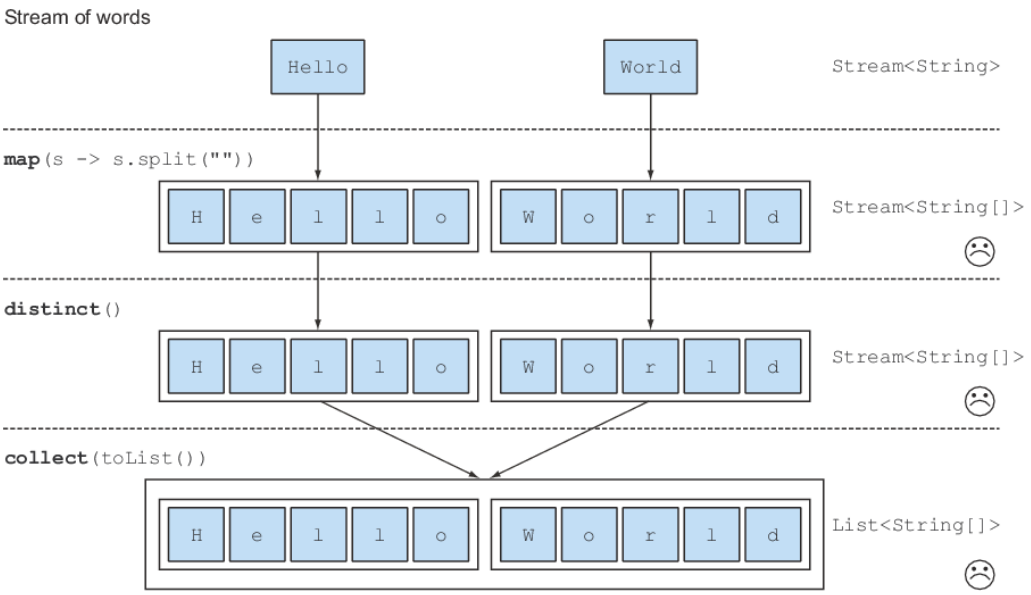


Figure 5.5 Incorrect use of `map` to find unique characters from a list of words

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(""))                    ❶
```

```
    .map(Arrays::stream)                    ❷
    .distinct()
    .collect(toList());
```

❶ Converts each word into an array of its individual letters
❷ 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, `List<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(word -> word.split(""))          ❶
       .flatMap(Arrays::stream)              ❷
       .distinct()
       .collect(toList());
```

❶ Converts each word into an array of its individual letters
❷ 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.

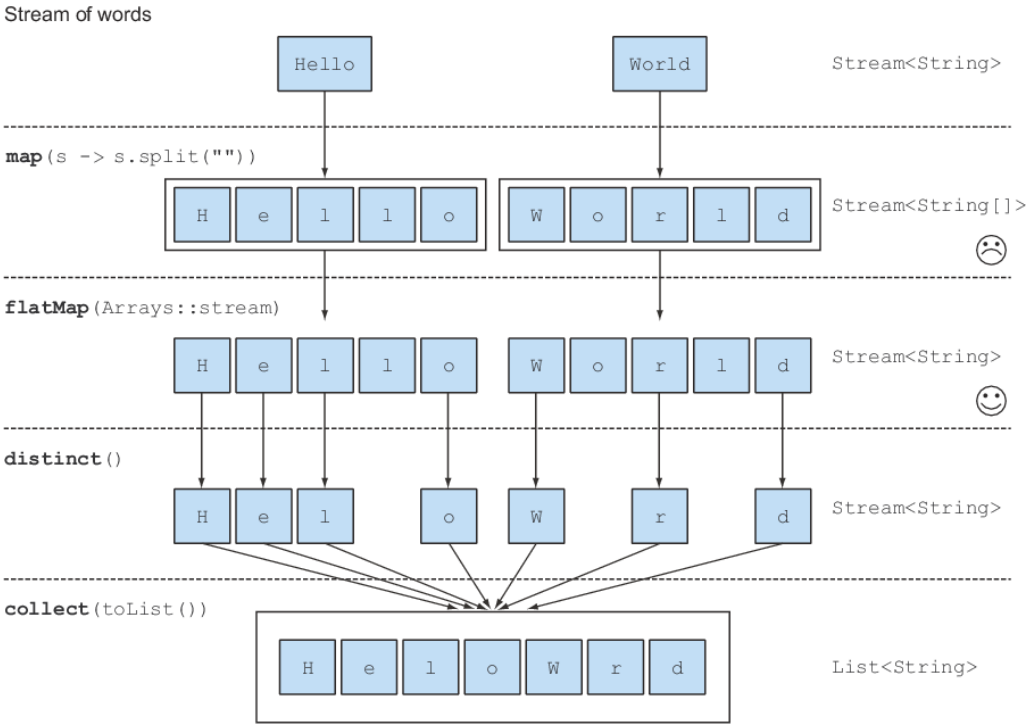


Figure 5.6 Using `flatMap` to find the unique characters from a list of words

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 11 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)].

5.4 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.

5.4.1 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.

5.4.2 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(dish -> dish.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.

5.4.3 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 11 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()                                                    (1)
    .ifPresent(dish -> System.out.println(dish.getName());        (2)
```

(1) Returns an Optional<Dish>.
(2) If a value is contained, it's printed; otherwise nothing happens.

5.4.4 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(n -> n * n)
             .filter(n -> n % 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.

5.5 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.

5.5.1 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.

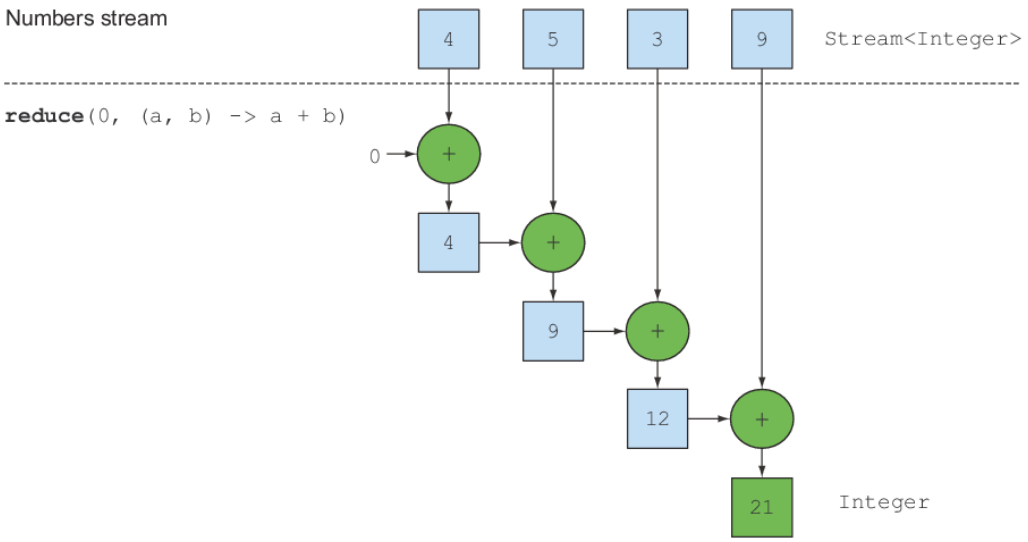


Figure 5.7 Using `reduce` to sum the numbers in a stream

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. From 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`.

5.5.2 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:

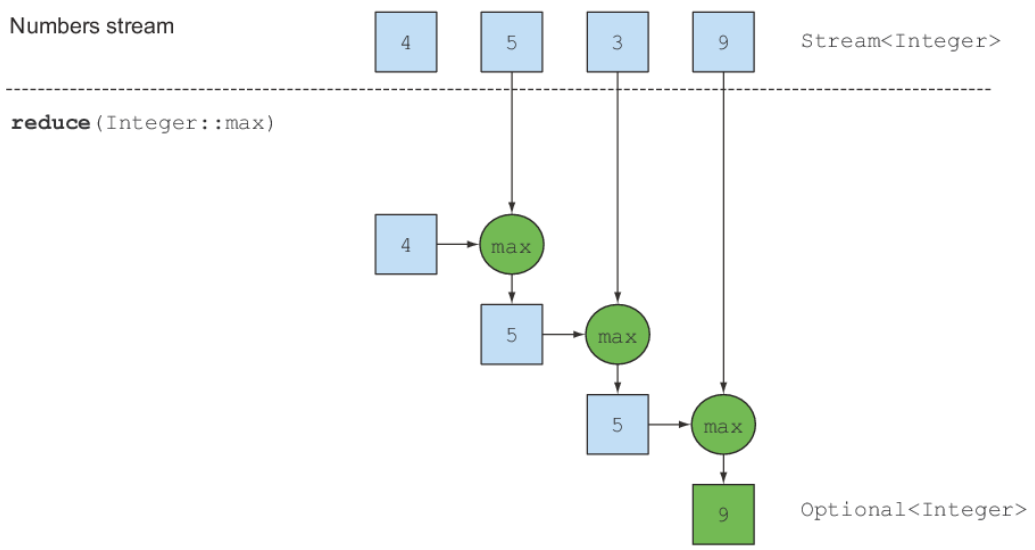


Figure 5.8 A reduce operation—calculating the maximum

```
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 definitely 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 and commutative 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 additional built-in methods such as `sum` and `max` are available 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>`		
`takeWhile`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`dropWhile`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`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>		
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		

5.6 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.

5.6.1 The domain: Traders and Transactions

Here's the domain you'll be working with, a list of `Trader`s and `Transaction`s:

```
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 +"}";
    }
}
```

5.6.2 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)      ❶
                .sorted(comparing(Transaction::getValue))                  ❷
                .collect(toList());                                        ❸
```

❶ Pass a predicate to filter to select transactions in year 2011.
❷ Sort them by using the value of the transaction.
❸ Collect all the elements of the resulting Stream into a List.

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

```
List<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())    (1)
                .distinct()                                               (2)
                .collect(toList());
```

(1) Extract the city from each trader associated with the transaction.
(2) 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)                              (1)
                .filter(trader -> trader.getCity().equals("Cambridge"))   (2)
                .distinct()                                               (3)
                .sorted(comparing(Trader::getName))                       (4)
                .collect(toList());
```

(1) Extract all traders from the transactions.
(2) Select only the traders from Cambridge.
(3) Make sure you don't have any duplicates.
(4) 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())    (1)
                .distinct()                                               (2)
                .sorted()                                                 (3)
                .reduce("", (n1, n2) -> n1 + n2);                         (4)
```

(1) Extract all the names of the traders as a Stream of Strings.
(2) Select only the unique names.
(3) Sort the names alphabetically.
(4) 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")); (1)
```

(1) 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())) (1)
            .map(Transaction::getValue)                               (2)
            .forEach(System.out::println);                            (3)
```

(1) Select the transactions where the traders live in Cambridge.
(2) Extract the values of these trades.
(3) Print each value.

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

```
Optional<Integer> highestValue =
    transactions.stream()
                .map(Transaction::getValue)                           (1)
                .reduce(Integer::max);                                (2)
```

(1) Extract the value of each transaction.
(2) 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);      (1)
```

(1) 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));
```

5.7 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.

5.7.1 Primitive stream specializations

Java 8 introduces three primitive specialized stream interfaces to tackle this issue, `IntStream`, `DoubleStream`, and `LongStream`, which 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()                          (1)
                   .mapToInt(Dish::getCalories)       (2)
                   .sum();
```

(1) **Returns a Stream<Dish>**
(2) **Returns an IntStream**

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);  (1)
Stream<Integer> stream = intStream.boxed();                       (2)
```

(1) Converting a Stream to a numeric stream
(2) 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);                  (1)
```

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

5.7.2 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)          (1)
                                 .filter(n -> n % 2 == 0);     (2)
System.out.println(evenNumbers.count());                       (3)
```

(1) Represents the range [1, 100].
(2) A stream of even numbers from 1 to 100.
(3) 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.

5.7.3 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.

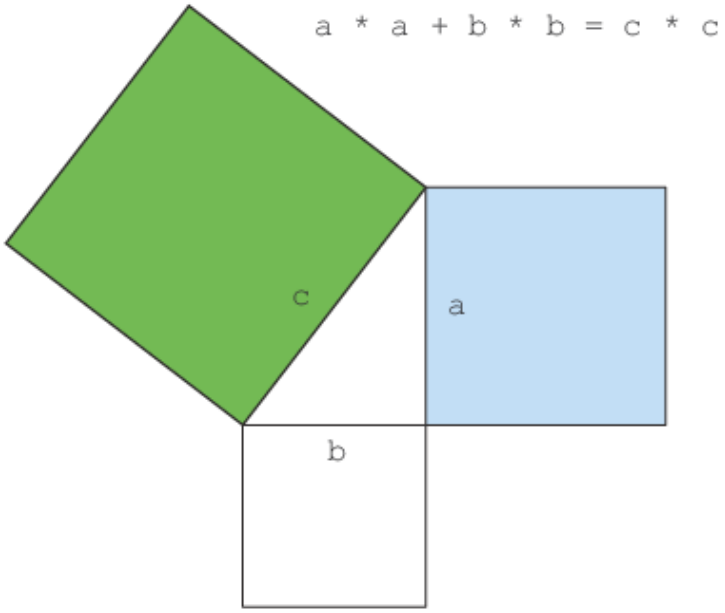


Figure 5.9 The Pythagorean theorem

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 a whole number; that is, its fractional part, which in Java can be expressed using `expr % 1.0`, is zero. 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)}) (1)
                       .filter(t -> t[2] % 1 == 0));                    (2)
```

(1) **Produce triples.**
(2) **The third element of the tuple must be an integer.**

5.8 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!

5.8.1 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("Modern ", "Java ", "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();
```

5.8.2 Stream from nullable

In Java 9, a new method was added which lets you create a stream from a nullable object. After playing with streams, you may have encountered a situation where you extracted an object that may be null and then needs to be converted into a stream (or an empty stream for null). For example, the method `System.getProperty` returns `null` if there is no property with the given key. To use it together with a stream you'd need to explicitly check for null as follows:

```
String homeValue = System.getProperty("home");
Stream<String> homeValueStream
    = homeValue == null ? Stream.empty() : Stream.of(value);
```

Using `Stream.ofNullable` you can rewrite this code more simply:

```
Stream<String> homeValueStream
    = Stream.ofNullable(System.getProperty("home"));
```

This pattern can be particularly handy in conjunction with `flatMap` and a stream of values that may include nullable objects:

```
Stream<String> values =
   Stream.of("config", "home", "user")
         .flatMap(key -> Stream.ofNullable(System.getProperty(key)));
```

5.8.3 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 `int`s into an `IntStream` as and then sum this `IntStream` to produce an `int` follows:

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();          (1)
```

(1) **The sum is 41.**

5.8.4 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())){   ❶
uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))        ❷
                   .distinct()                                             ❸
                   .count();                                               ❹
}
catch(IOException e){
                                                                           ❺
}
```

❶ Streams are autoclosable.
❷ Generate a stream of words.
❸ Remove duplicates.
❹ Count the number of unique words.
❺ 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. This call is surrounded by a try/catch block because the source of the Stream is an I/O resource. In fact, the call `Files.lines` will open an I/O resource, which needs to be closed to avoid a leak. In the past, you'd need an explicit `finally` block to do this. Conveniently, the `Stream` interface implements the interface `AutoCloseable`. This means that the management of the resource is handled for you within the `try` block. Once you have a stream of lines, you can 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`.

5.8.5 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 `UnaryOperator<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....

In Java 9, the `iterate` method was enhanced with support for a predicate. For example, you can generate numbers starting at 0 but stop the iteration once the number are greater than 100:

```
IntStream.iterate(0, n -> n < 100, n -> n + 4)
         .forEach(System.out::println);
```

The `iterate` method takes a predicate as its second argument that tells you when to continue iterating up until. Note that you may think that you can the `filter` operation to achieve the same result:

```
IntStream.iterate(0, n -> n + 4)
         .filter(n -> n < 100)
         .forEach(System.out::println);
```

Unfortunately that is not the case. In fact, this code would not terminate! The reason is that there's no way to know in the filter that the numbers continue to increase so it keeps on filtering them infinitely! You could solve the problem by using `takeWhile` which would short-circuit the stream:

```
IntStream.iterate(0, n -> n + 4)
         .takeWhile(n -> n < 100)
         .forEach(System.out::println);
```

However, you have to admit that `iterate` with a predicate is a bit more concise!

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 whether 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 the stateful `IntSupplier` for Fibonacci below is shown 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 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);
```

The above code creates 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 contains an infinite number of elements!

5.9 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`, `takeWhile` (Java 9), `dropWhile` (Java 9), `skip`, and `limit` methods.
- The method `takeWhile` and `dropWhile` are more efficient than filter when you know that the source is sorted.
- 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 an infinite number of elements (e.g. all possible Strings). This is possible because the elements of a stream are only produced *on demand*. You can get a finite stream from an infinite stream using methods such as `limit`.

6

Collecting data with streams

This chapter covers

- **Creating and using a collector with the `Collectors` class**
- **Reducing streams of data to a single value**
- **Summarization as a special case of reduction**
- **Grouping and partitioning data**
- **Developing your own custom collectors**

You learned in the previous chapter that streams help you process collections with database-like operations. You can view Java 8 streams as fancy lazy iterators of sets of data. They support two types of operations: intermediate operations such as `filter` or `map` and terminal operations such as `count`, `findFirst`, `forEach`, and `reduce`. Intermediate operations can be chained to convert a stream into another stream. These operations don't consume from a stream; their purpose is to set up a pipeline of streams. By contrast, terminal operations *do* consume from a stream—to produce a final result (for example, returning the largest element in a stream). They can often shorten computations by optimizing the pipeline of a stream.

We already used the `collect` terminal operation on streams in chapters 4 and 5, but we employed it there mainly to combine all the elements of a `Stream` into a `List`. In this chapter, you'll discover that `collect` is a reduction operation, just like `reduce`, that takes as argument various recipes for accumulating the elements of a stream into a summary result. These recipes are defined by a new `Collector` interface, so it's important to distinguish `Collection`, `Collector`, and `collect`!

Here are some example queries of what you'll be able to do using `collect` and collectors:

- Group a list of transactions by currency to obtain the sum of the values of all transactions with that currency (returning a `Map<Currency, Integer>`)

- Partition a list of transactions into two groups: expensive and not expensive (returning a `Map<Boolean, List<Transaction>>`)
- Create multilevel groupings such as grouping transactions by cities and then further categorizing by whether they're expensive or not (returning a `Map<String, Map<Boolean, List<Transaction>>>`)

Excited? Great, let's start by exploring an example that benefits from collectors. Imagine a scenario where you have a `List` of `Transaction`s, and you want to group them based on their nominal currency. In pre-lambda Java, even a simple use case like this is cumbersome to implement, as shown in the following listing.

Listing 6.1 Grouping transactions by currency in imperative style

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
                                                new HashMap<>();  (1)
for (Transaction transaction : transactions) {                    (2)
    Currency currency = transaction.getCurrency();                (3)
    List<Transaction> transactionsForCurrency =
                                    transactionsByCurrencies.get(currency);
    if (transactionsForCurrency == null) {                        (4)
        transactionsForCurrency = new ArrayList<>();
        transactionsByCurrencies
                            .put(currency, transactionsForCurrency);
    }
    transactionsForCurrency.add(transaction);                     (5)
}
```

(1) Create the Map where the grouped transaction will be accumulated.
(2) Iterate the List of Transactions.
(3) Extract the Transaction's currency.
(4) If there's no entry in the grouping Map for this currency, create it.
(5) Add the currently traversed Transaction to the List of Transactions with the same currency.

If you're an experienced Java developer, you'll probably feel comfortable writing something like this, but you have to admit that it's a lot of code for such a simple task. Even worse, this is probably harder to read than to write! The purpose of the code isn't immediately evident at first glance, even though it can be expressed in a straightforward manner in plain English: "Group a list of transactions by their currency." As you'll learn in this chapter, you can achieve exactly the same result with a single statement by using a more general `Collector` parameter to the `collect` method on `Stream` rather than the `toList` special case used in the previous chapter:

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
        transactions.stream().collect(groupingBy(Transaction::getCurrency));
```

The comparison is quite embarrassing, isn't it?

6.1 Collectors in a nutshell

The previous example clearly shows one of the main advantages of functional-style programming over an imperative approach: you just have to formulate the result you want to obtain the "what" and not the steps you need to perform to obtain it—the "how." In the previous example, the argument passed to the `collect` method is an implementation of the `Collector` interface, which is a recipe for how to build a summary of the elements in the `Stream`. In the previous chapter, the `toList` recipe just said "Make a list of each element in turn"; in this example, the `groupingBy` recipe says "Make a `Map` whose keys are (currency) buckets and whose values are a list of elements in those buckets."

The difference between the imperative and functional versions of this example is even more pronounced if you perform multilevel groupings: in this case the imperative code quickly becomes harder to read, maintain, and modify due to the number of deeply nested loops and conditions required. In comparison, the functional-style version, as you'll discover in section 6.3, can be easily enhanced with an additional collector.

6.1.1 Collectors as advanced reductions

This last observation brings up another typical benefit of a well-designed functional API: its higher degree of composability and reusability. Collectors are extremely useful because they provide a concise yet flexible way to define the criteria that `collect` uses to produce the resulting collection. More specifically, invoking the `collect` method on a stream triggers a reduction operation (parameterized by a `Collector`) on the elements of the stream itself. This *reduction operation*, illustrated in figure 6.1, internally does for you what you had to code imperatively in listing 6.1. It traverses each element of the stream and lets the `Collector` process them.

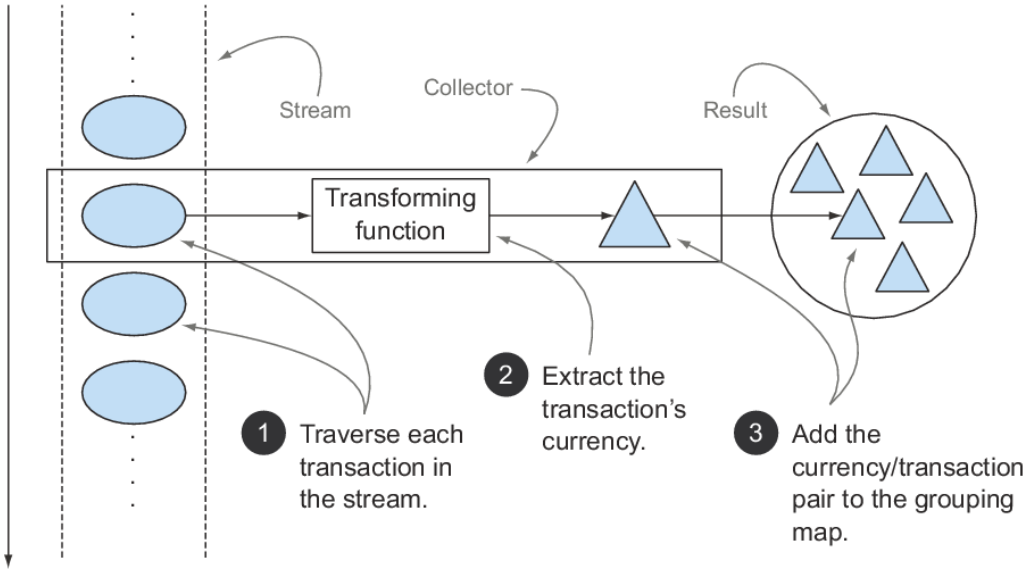


Figure 6.1 The reduction process grouping the transactions by currency

Typically, the `Collector` applies a transforming function to the element (quite often this is the identity transformation, which has no effect, for example, as in `toList`), and accumulates the result in a data structure that forms the final output of this process. For instance, in our transaction-grouping example shown previously, the transformation function extracts the currency from each transaction, and subsequently the transaction itself is accumulated in the resulting `Map`, using the currency as key.

The implementation of the methods of the `Collector` interface defines how to perform a reduction operation on a stream, such as the one in our currency example. We investigate how to create customized collectors in sections 6.5 and 6.6. But the `Collectors` utility class provides lots of static factory methods to conveniently create an instance of the most common collectors that are ready to use. The most straightforward and frequently used collector is the `toList` static method, which gathers all the elements of a stream into a `List`:

```
List<Transaction> transactions =
    transactionStream.collect(Collectors.toList());
```

6.1.2 Predefined collectors

In the rest of this chapter, we mainly explore the features of the predefined collectors, those that can be created from the factory methods (such as `groupingBy`) provided by the `Collectors` class. These offer three main functionalities:

- Reducing and summarizing stream elements to a single value

- Grouping elements
- Partitioning elements

We start with collectors that allow you to reduce and summarize. These are handy in a variety of use cases such as finding the total amount of the transacted values in the list of transactions in the previous example.

You'll then see how to group the elements of a stream, generalizing the previous example to multiple levels of grouping or combining different collectors to apply further reduction operations on each of the resulting subgroups. We'll also describe *partitioning* as a special case of grouping, using a predicate, a one-argument function returning a boolean, as a grouping function.

At the end of section 6.4 you'll find a table summarizing all the predefined collectors explored in this chapter. Finally, in section 6.5 you'll learn more about the `Collector` interface before you explore (section 6.6) how you can create your own custom collectors to be used in the cases not covered by the factory methods of the `Collectors` class.

6.2 Reducing and summarizing

To illustrate the range of possible collector instances that can be created from the `Collectors` factory class, we'll reuse the domain we introduced in the previous chapter: a menu consisting of a list of delicious dishes!

As you just learned, collectors (the parameters to the `Stream` method `collect`) are typically used in cases where it's necessary to reorganize the stream's items into a collection. But more generally, they can be used every time you want to combine all the items in the stream into a single result. This result can be of any type, as complex as a multilevel map representing a tree or as simple as a single integer—perhaps representing the sum of all the calories in the menu. We'll look at both of these result types: single integers in section 6.2.2 and multilevel grouping in section 6.3.1.

As a first simple example, let's count the number of dishes in the menu, using the collector returned by the `counting` factory method:

```
long howManyDishes = menu.stream().collect(Collectors.counting());
```

You can write this far more directly as

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

but the `counting` collector can be especially useful when used in combination with other collectors, as we demonstrate later.

In the rest of this chapter, we assume that you've imported all the static factory methods of the `Collectors` class with

```
import static java.util.stream.Collectors.*;
```

so you can write `counting()` instead of `Collectors.counting()` and so on.

Let's continue exploring simple predefined collectors by looking at how you can find the maximum and minimum values in a stream.

6.2.1 Finding maximum and minimum in a stream of values

Suppose you want to find the highest-calorie dish in the menu. You can use two collectors, `Collectors.maxBy` and `Collectors.minBy`, to calculate the maximum or minimum value in a stream. These two collectors take a `Comparator` as argument to compare the elements in the stream. Here you create a `Comparator` comparing dishes based on their calorie content and pass it to `Collectors.maxBy`:

```
Comparator<Dish> dishCaloriesComparator =
    Comparator.comparingInt(Dish::getCalories);
Optional<Dish> mostCalorieDish =
    menu.stream()
        .collect(maxBy(dishCaloriesComparator));
```

You may wonder what the `Optional<Dish>` is about. To answer this we need to ask the question "What if `menu` were empty?" There's no dish to return! Java 8 introduces `Optional`, which is a container that may or may not contain a value. Here it perfectly represents the idea that there may or may not be a dish returned. We briefly mentioned it in chapter 5 when you encountered the method `findAny`. Don't worry about it for now; we devote chapter 11 to the study of `Optional<T>` and its operations.

Another common reduction operation that returns a single value is to sum the values of a numeric field of the objects in a stream. Alternatively, you may want to average the values. Such operations are called *summarization* operations. Let's see how you can express them using collectors.

6.2.2 Summarization

The `Collectors` class provides a specific factory method for summing: `Collectors.summingInt`. It accepts a function that maps an object into the `int` that has to be summed and returns a collector that, when passed to the usual `collect` method, performs the requested summarization. So, for instance, you can find the total number of calories in your menu list with

```
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

Here the collection process proceeds as illustrated in figure 6.2. While traversing the stream each dish is mapped into its number of calories, and this number is added to an accumulator starting from an initial value (in this case the value is `0`).

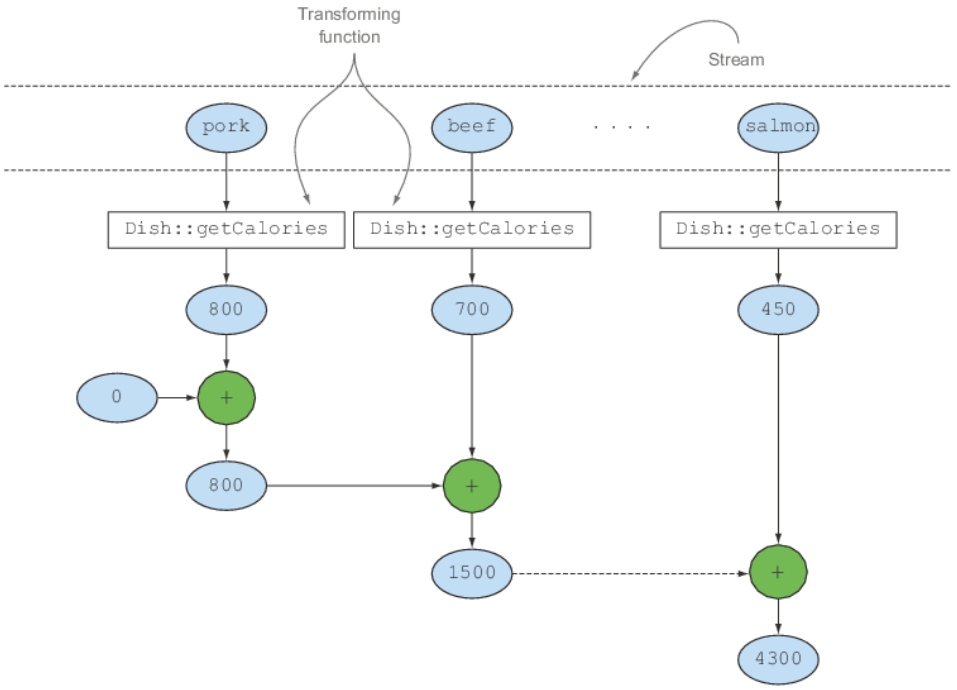


Figure 6.2 The aggregation process of the `summingInt` collector

The `Collectors.summingLong` and `Collectors.summingDouble` methods behave exactly the same way and can be used where the field to be summed is respectively a `long` or a `double`.

But there's more to summarization than mere summing; also available is a `Collectors.averagingInt`, together with its `averagingLong` and `averagingDouble` counterparts, to calculate the average of the same set of numeric values:

```
double avgCalories =
    menu.stream().collect(averagingInt(Dish::getCalories));
```

So far, you've seen how to use collectors to count the elements in a stream, find the maximum and minimum values of a numeric property of those elements, and calculate their sum and average. Quite often, though, you may want to retrieve two or more of these results, and possibly you'd like to do it in a single operation. In this case, you can use the collector returned by the `summarizingInt` factory method. For example, you can count the elements in the menu and obtain the sum, average, maximum, and minimum of the calories contained in each dish with a single `summarizing` operation:

```
IntSummaryStatistics menuStatistics =
        menu.stream().collect(summarizingInt(Dish::getCalories));
```

This collector gathers all that information in a class called `IntSummaryStatistics` that provides convenient getter methods to access the results. Printing the `menu-Statistic` object produces the following output:

```
IntSummaryStatistics{count=9, sum=4300, min=120,
                     average=477.777778, max=800}
```

As usual, there are corresponding `summarizingLong` and `summarizingDouble` factory methods with associated types `LongSummaryStatistics` and `DoubleSummaryStatistics`; these are used when the property to be collected is a primitive-type `long` or a `double`.

6.2.3 Joining Strings

The collector returned by the `joining` factory method concatenates into a single string all strings resulting from invoking the `toString` method on each object in the stream. This means you can concatenate the names of all the dishes in the menu as follows:

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

Note that `joining` internally makes use of a `StringBuilder` to append the generated strings into one. Also note that if the `Dish` class had a `toString` method returning the dish's name, you'd obtain the same result without needing to map over the original stream with a function extracting the name from each dish:

```
String shortMenu = menu.stream().collect(joining());
```

Both produce the following string,

```
porkbeefchickenfrench friesriceseason fruitpizzaprawnssalmon
```

which isn't very readable. Fortunately, the `joining` factory method is overloaded, with one of its overloaded variants taking a string used to delimit two consecutive elements, so you can obtain a comma-separated list of the dishes' names with

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
```

which, as expected, will generate

```
pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon
```

Until now, we've explored various collectors that reduce a stream to a single value. In the next section, we demonstrate how all the reduction processes of this form are special cases of the more general reduction collector provided by the `Collectors.reducing` factory method.

6.2.4 Generalized summarization with reduction

All the collectors we've discussed so far are, in reality, only convenient specializations of a reduction process that can be defined using the `reducing` factory method. The `Collectors.reducing` factory method is a generalization of all of them. The special cases

discussed earlier are arguably provided only for programmer convenience. (But remember that programmer convenience and readability are of prime importance!) For instance, it's *possible* to calculate the total calories in your menu with a collector created from the `reducing` method as follows:

```
int totalCalories = menu.stream().collect(reducing(
                                    0, Dish::getCalories, (i, j) -> i + j));
```

It takes three arguments:

- The first argument is the starting value of the reduction operation and will also be the value returned in the case of a stream with no elements, so clearly `0` is the appropriate value in the case of a numeric sum.
- The second argument is the same function you used in section 6.2.2 to transform a dish into an `int` representing its calorie content.
- The third argument is a `BinaryOperator` that aggregates two items into a single value of the same type. Here, it just sums two `int`s.

Similarly, you could find the highest-calorie dish using the one-argument version of `reducing` as follows:

```
Optional<Dish> mostCalorieDish =
    menu.stream().collect(reducing(
        (d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2));
```

You can think of the collector created with the one-argument `reducing` factory method as a particular case of the three-argument method, which uses the first item in the stream as a starting point and an *identity function* (that is, a function doing nothing more than returning its input argument as is) as a transformation function. This also implies that the one-argument `reducing` collector won't have any starting point when passed to the `collect` method of an empty stream and, as we explained in section 6.2.1, for this reason it returns an `Optional<Dish>` object.

Collect vs. reduce

We've discussed reductions a lot in the previous chapter and this one. You may naturally wonder what the differences between the `collect` and `reduce` methods of the `Stream` interface are, because often you can obtain the same results using either method. For instance, you can achieve what is done by the `toList` `Collector` using the `reduce` method as follows:

```
Stream<Integer> stream = Arrays.asList(1, 2, 3, 4, 5, 6).stream();
List<Integer> numbers = stream.reduce(
                              new ArrayList<Integer>(),
                              (List<Integer> l, Integer e) -> {
                                        l.add(e);
                                        return l; },
                              (List<Integer> l1, List<Integer> l2) -> {
                                        l1.addAll(l2);
                                        return l1; });
```

This solution has two problems: a semantic one and a practical one. The semantic problem lies in the fact that the `reduce` method is meant to combine two values and produce a new one; it's an immutable reduction. In contrast, the `collect` method is designed to mutate a container to accumulate the result it's supposed to produce. This means that the previous snippet of code is misusing the `reduce` method because it's mutating in place the `List` used as accumulator. As you'll see in more detail in the next chapter, using the `reduce` method with the wrong semantic is also the cause of a practical problem: this reduction process can't work in parallel because the concurrent modification of the same data structure operated by multiple threads can corrupt the `List` itself. In this case, if you want thread safety, you'll need to allocate a new `List` every time, which would impair performance by object allocation. This is the main reason why the `collect` method is useful for expressing reduction working on a mutable container but crucially in a parallel-friendly way, as you'll learn later in the chapter.

COLLECTION FRAMEWORK FLEXIBILITY: DOING THE SAME OPERATION IN DIFFERENT WAYS

You can further simplify the previous sum example using the `reducing` collector by using a reference to the `sum` method of the `Integer` class instead of the lambda expression you used to encode the same operation. This results in the following:

```
int totalCalories = menu.stream().collect(reducing(0,          ❶
                                Dish::getCalories,             ❷
                                Integer::sum));                ❸
```

❶ **Initial value**
❷ **Transformation function**
❸ **Aggregating function**

Logically, this reduction operation proceeds as shown in figure 6.3, where an accumulator, initialized with a starting value, is iteratively combined, using an aggregating function, with the result of the application of the transforming function on each element of the stream.

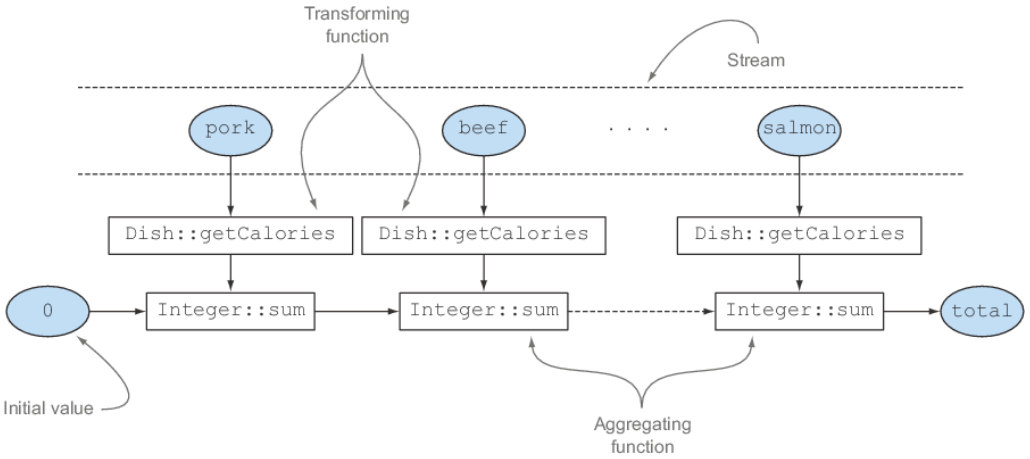


Figure 6.3 The reduction process calculating the total number of calories in the menu

The `counting` collector we mentioned at the beginning of section 6.2 is, in reality, similarly implemented using the three-argument `reducing` factory method. It transforms each element in the stream to an object of type `Long` with value `1` and then sums all these ones:

```
public static <T> Collector<T, ?, Long> counting() {
    return reducing(0L, e -> 1L, Long::sum);
}
```

Use of the generic `?` wildcard

In the code snippet just shown, you probably noticed the `?` wildcard, used as the second generic type in the signature of the collector returned by the `counting` factory method. You should already be familiar with this notation, especially if you use the Java Collection Framework quite frequently. But here it means only that the type of the collector's accumulator is unknown, or in other words, the accumulator itself can be of any type. We used it here to exactly report the signature of the method as originally defined in the `Collectors` class, but in the rest of the chapter we avoid any wildcard notation to keep the discussion as simple as possible.

We already observed in chapter 5 that there's another way to perform the same operation without using a collector—by mapping the stream of dishes into the number of calories of each dish and then reducing this resulting stream with the same method reference used in the previous version:

```
int totalCalories =
    menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();
```

Note that, like any one-argument `reduce` operation on a stream, the invocation `reduce(Integer::sum)` doesn't return an `int` but an `Optional<Integer>` to manage in a null-safe way the case of a reduction operation over an empty stream. Here you just extract the

value inside the `Optional` object using its `get` method. Note that in this case using the `get` method is safe only because you're sure that the stream of dishes isn't empty. In general, as you'll learn in chapter 10, it's safer to unwrap the value eventually contained in an `Optional` using a method that also allows you to provide a default, such as `orElse` or `orElseGet`. Finally, and even more concisely, you can achieve the same result by mapping the stream to an `IntStream` and then invoking the `sum` method on it:

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

CHOOSING THE BEST SOLUTION FOR YOUR SITUATION

Once again, this demonstrates how functional programming in general (and the new API based on functional-style principles added to the `Collections` framework in Java 8 in particular) often provides multiple ways to perform the same operation. This example also shows that collectors are somewhat more complex to use than the methods directly available on the `Streams` interface, but in exchange they offer higher levels of abstraction and generalization and are more reusable and customizable.

Our suggestion is to explore the largest number of solutions possible for the problem at hand, but always choose the most specialized one that's general enough to solve it. This is often the best decision for both readability and performance reasons. For instance, to calculate the total calories in our menu, we'd prefer the last solution (using `IntStream`) because it's the most concise and likely also the most readable one. At the same time, it's also the one that performs best, because `IntStream` lets us avoid all the *auto-unboxing* operations, or implicit conversions from `Integer` to `int`, that are useless in this case.

Next, take the time to test your understanding of how `reducing` can be used as a generalization of other collectors by working through the exercise in Quiz 6.1.

Quiz 6.1: Joining strings with reducing

Which of the following statements using the `reducing` collector are valid replacements for this `joining` collector (as used in section 6.2.3)?

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
 1)  String shortMenu = menu.stream().map(Dish::getName)
       .collect( reducing( (s1, s2) -> s1 + s2 ) ).get();
 2)  String shortMenu = menu.stream()
       .collect( reducing( (d1, d2) -> d1.getName() + d2.getName() ) ).get();
 3)  String shortMenu = menu.stream()
       .collect( reducing( "", Dish::getName, (s1, s2) -> s1 + s2 ) );
```

Answer:

Statements 1 and 3 are valid, whereas 2 doesn't compile.

1) **This converts each dish in its name, as done by the original statement using the `joining` collector, and then reduces the resulting stream of strings using a `String` as accumulator and appending to it the names of the dishes one by one.**

2) This doesn't compile because the one argument that `reducing` accepts is a `BinaryOperator<T>` that's a `BiFunction<T,T,T>`. This means that it wants a function taking two arguments and returns a value of the same type, but the lambda expression used there has two dishes as arguments but returns a string.
3) This starts the reduction process with an empty string as the accumulator, and when traversing the stream of dishes it converts each dish to its name and appends this name to the accumulator. Note that, as we mentioned, `reducing` doesn't need the three arguments to return an `Optional` because in the case of an empty stream it can return a more meaningful value, which is the empty string used as the initial accumulator value.

Note that even though statements 1 and 3 are valid replacements for the `joining` collector, they've been used here to demonstrate how the `reducing` one can be seen, at least conceptually, as a generalization of all other collectors discussed in this chapter. Nevertheless, for all practical purposes we always suggest using the `joining` collector for both readability and performance reasons.

6.3 Grouping

A common database operation is to group items in a set, based on one or more properties. As you saw in the earlier transactions-currency-grouping example, this operation can be cumbersome, verbose, and error prone when implemented with an imperative style. But it can be easily translated in a single, very readable statement by rewriting it in a more functional style as encouraged by Java 8. As a second example of how this feature works, suppose you want to classify the dishes in the menu according to their type, putting the ones containing meat in a group, the ones with fish in another group, and all others in a third group. You can easily perform this task using a collector returned by the `Collectors.groupingBy` factory method as follows:

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

This will result in the following `Map`:

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

Here, you pass to the `groupingBy` method a `Function` (expressed in the form of a method reference) extracting the corresponding `Dish.Type` for each `Dish` in the stream. We call this `Function` a *classification* function because it's used to classify the elements of the stream into different groups. The result of this grouping operation, shown in figure 6.4, is a `Map` having as map key the value returned by the classification function and as corresponding map value a list of all the items in the stream having that classified value. In the menu-classification example a key is the type of dish, and its value is a list containing all the dishes of that type.

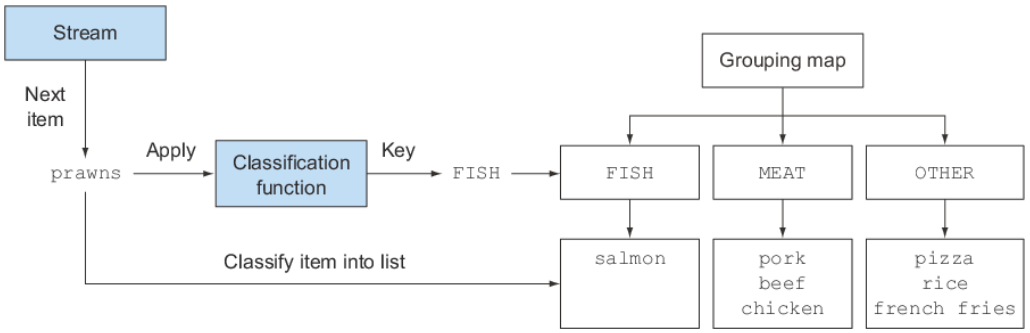


Figure 6.4 Classification of an item in the stream during the grouping process

But it isn't always possible to use a method reference as a classification function, because you may wish to classify using something more complex than a simple property accessor. For instance, you could decide to classify as "diet" all dishes with 400 calories or fewer, set to "normal" the dishes having between 400 and 700 calories, and set to "fat" the ones with more than 700 calories. Because the author of the `Dish` class unhelpfully didn't provide such an operation as a method, you can't use a method reference in this case, but you can express this logic in a lambda expression:

```
public enum CaloricLevel { DIET, NORMAL, FAT }
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream().collect(
         groupingBy(dish -> {
                if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
        else return CaloricLevel.FAT;
         } ));
```

Now you've seen how to group the dishes in the menu, both by their type and by calories, but it could be also quite common that you may need to further manipulate the results of the original grouping and in the next section we will show how you achieve this.

6.3.1 Manipulating grouped elements

It could happen very frequently that after having performed a grouping operation you can have the necessity of manipulating the elements in each resulting group. Suppose, for example, that you want to filter only the caloric dishes, let's say the ones with more than 500 calories. You may argue that in this case you could just apply this filtering predicate just before the grouping like in:

```
Map<Dish.Type, List<Dish>> caloricDishesByType =
                    menu.stream().filter(dish -> dish.getCalories() > 500)
                                 .collect(groupingBy(Dish::getType));
```

This solution works but has a possibility relevant drawback. If you try to use it on the dishes in our menu you will obtain a Map like this:

```
{OTHER=[french fries, pizza], MEAT=[pork, beef]}
```

Do you see the problem there? Since there is no dish of type FISH satisfying our filtering predicate, that key totally disappeared from the resulting map. To workaround this problem the `Collectors` class overloads the `groupingBy` factory method, with one variant also taking a second argument of type `Collector` along with the usual classification function. In this way it is possible to move the filtering predicate inside this second `Collector` as follows:

```
Map<Dish.Type, List<Dish>> caloricDishesByType =
      menu.stream()
          .collect(groupingBy(Dish::getType,
                   filtering(dish -> dish.getCalories() > 500, toList())));
```

The `filtering` method is another static factory method of the `Collectors` class accepting a `Predicate` to filter the elements in each group and a further `Collector` that is used to regroup the filtered elements. In this way the resulting Map will keep an entry also for the FISH type even if it maps an empty List.

```
{OTHER=[french fries, pizza], MEAT=[pork, beef], FISH=[]}
```

Another even more common way in which it could be useful to manipulate the grouped elements is transforming them through a mapping function. To this purpose, similarly to what have just seen for the `filtering Collector`, the `Collectors` class provides another `Collector` through the `mapping` method that accepts a mapping function and another `Collector` used to gather the elements resulting from the application of that function to each of them. By using it you can for instance convert each Dish in the groups into their respective names in this way:

```
Map<Dish.Type, List<String>> dishNamesByType =
      menu.stream()
          .collect(groupingBy(Dish::getType,
                   mapping(Dish::getName, toList())));
```

Note that in this case each group in the resulting Map is a List of Strings rather than one of Dishes as it was in the former examples. There is also a third `Collector` that you could use in combination with the `groupingBy` one and that allows to peform a flatMap transformation instead of a plain map. To demonstrate how this works let's suppose that we have a Map associating to each Dish a list of tags as it follows:

```
Map<String, List<String>> dishTags = new HashMap<>();
dishTags.put("pork", asList("greasy", "salty"));
dishTags.put("beef", asList("salty", "roasted"));
dishTags.put("chicken", asList("fried", "crisp"));
dishTags.put("french fries", asList("greasy", "fried"));
dishTags.put("rice", asList("light", "natural"));
dishTags.put("season fruit", asList("fresh", "natural"));
dishTags.put("pizza", asList("tasty", "salty"));
```

```
dishTags.put("prawns", asList("tasty", "roasted"));
dishTags.put("salmon", asList("delicious", "fresh"));
```

In case you are required to extract these tags for each group of type of dishes you can easily achieve this using the `flatMapping Collector`.

```
Map<Dish.Type, Set<String>> dishNamesByType =
   menu.stream()
      .collect(groupingBy(Dish::getType,
               flatMapping(dish -> dishTags.get( dish.getName() ).stream(),
                           toSet())));
```

Here for each Dish we are obtaining a List of tag, so analogously to what we have already seen in the former chapter, we need to perform a flatMap in order to flatten the resulting two level list into a single one. Also note that this time we collected the result of the flatMapping operations executed in each group into a Set instead of using a List as we did before, in order to avoid repetitions of same tags associated to more than one Dish in the same type. The Map resulting from this operation is then the following:

```
{MEAT=[salty, greasy, roasted, fried, crisp], FISH=[roasted, tasty, fresh, delicious],
       OTHER=[salty, greasy, natural, light, tasty, fresh, fried]}
```

Until this point we only used a single criteria to group the dishes in the menu, for instance by their type or by calories, but what if you want to use more than one criteria at the same time? Grouping is powerful because it composes effectively. Let's see how to do this.

6.3.2 Multilevel grouping

The 2 arguments `Collectors.groupingBy` factory method that we used in former section to manipulate the elements in the groups resulting from the grouping operation can be used also to perform a two-level grouping. To achieve this you can pass to it a second inner `groupingBy` to the outer `groupingBy`, defining a second-level criterion to classify the stream's items, as shown in the next listing.

Listing 6.2 Multilevel grouping

```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
menu.stream().collect(
      groupingBy(Dish::getType,                          (1)
         groupingBy(dish -> {                            (2)
            if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
           else return CaloricLevel.FAT;
          } )
      )
);
```

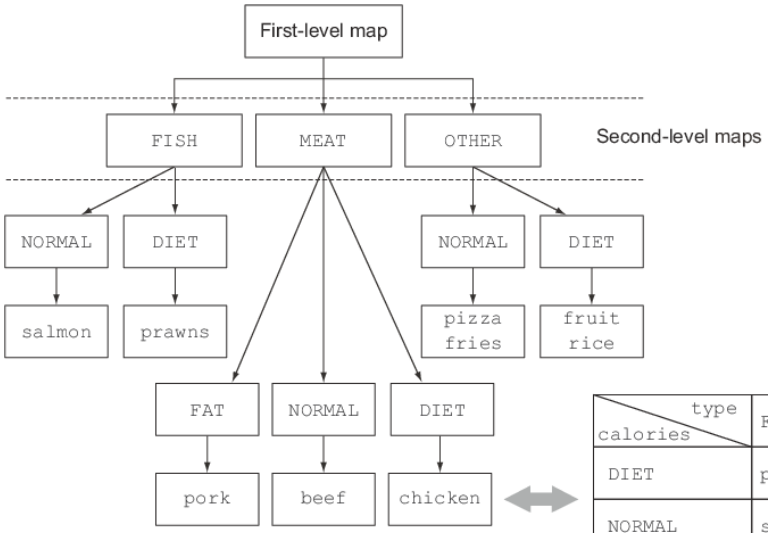
(1) First-level classification function
(2) Second-level classification function

The result of this two-level grouping is a two-level `Map` like the following:

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

Here the outer `Map` has as keys the values generated by the first-level classification function: "fish, meat, other." The values of this `Map` are in turn other `Map`s, having as keys the values generated by the second-level classification function: "normal, diet, or fat." Finally, the second-level `Map`s have as values the `List` of the elements in the stream returning the corresponding first- and second-level key values when applied respectively to the first and second classification functions: "salmon, pizza, etc." This multilevel grouping operation can be extended to any number of levels, and an *n*-level grouping has as a result an *n*-level `Map` modeling an *n*-level tree structure.

Figure 6.5 shows how this structure is also equivalent to an *n*-dimensional table, highlighting the classification purpose of the grouping operation.



type / calories	FISH	MEAT	OTHER
DIET	prawns	chicken	pizza fries
NORMAL	salmon	beef	fruit rice
FAT		pork	

Figure 6.5 Equivalence between *n*-level nested map and *n*-dimensional classification table

In general, it helps to think that `groupingBy` works in terms of "buckets." The first `groupingBy` creates a bucket for each key. You then collect the elements in each bucket with the downstream collector and so on to achieve *n*-level groupings!

6.3.3 Collecting data in subgroups

In the previous section, you saw that it's possible to pass a second `groupingBy` collector to the outer one to achieve a multilevel grouping. But more generally, the second collector passed to the first `groupingBy` can be any type of collector, not just another `groupingBy`. For instance, it's possible to count the number of `Dish`es in the menu for each type, by passing the `counting` collector as a second argument to the `groupingBy` collector:

```
Map<Dish.Type, Long> typesCount = menu.stream().collect(
                   groupingBy(Dish::getType, counting()));
```

The result is the following `Map`:

```
{MEAT=3, FISH=2, OTHER=4}
```

Also note that the regular one-argument `groupingBy(f)`, where `f` is the classification function, is in reality just shorthand for `groupingBy(f, toList())`.

To give another example, you could rework the collector you already used to find the highest-calorie dish in the menu to achieve a similar result, but now classified by the *type* of dish:

```
Map<Dish.Type, Optional<Dish>> mostCaloricByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
                            maxBy(comparingInt(Dish::getCalories))));
```

The result of this grouping is then clearly a `Map`, having as keys the available types of `Dish`es and as values the `Optional<Dish>`, wrapping the corresponding highest-calorie `Dish` for a given type:

```
{FISH=Optional[salmon], OTHER=Optional[pizza], MEAT=Optional[pork]}
```

> **NOTE** The values in this `Map` are `Optional`s because this is the resulting type of the collector generated by the `maxBy` factory method, but in reality if there's no `Dish` in the menu for a given type, that type won't have an `Optional.empty()` as value; it won't be present at all as a key in the `Map`. The `groupingBy` collector lazily adds a new key in the grouping `Map` only the first time it finds an element in the stream, producing that key when applying on it the grouping criteria being used. This means that in this case, the `Optional` wrapper isn't very useful, because it's not modeling a value that could be eventually absent but is there incidentally, only because this is the type returned by the reducing collector.

ADAPTING THE COLLECTOR RESULT TO A DIFFERENT TYPE

Because the `Optional`s wrapping all the values in the `Map` resulting from the last grouping operation aren't very useful in this case, you may want to get rid of them. To achieve this, or more generally, to adapt the result returned by a collector to a different type, you could use the collector returned by the `Collectors.collectingAndThen` factory method, as shown in the following listing.

Listing 6.3 Finding the highest-calorie `Dish` in each subgroup

```
Map<Dish.Type, Dish> mostCaloricByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,                    (1)
                 collectingAndThen(
                   maxBy(comparingInt(Dish::getCalories)),    (2)
                 Optional::get)));                            (3)
```

(1) Classification function
(2) Wrapped collector
(3) Transformation function

This factory method takes two arguments, the collector to be adapted and a transformation function, and returns another collector. This additional collector acts as a wrapper for the old one and maps the value it returns using the transformation function as the last step of the `collect` operation. In this case, the wrapped collector is the one created with `maxBy`, and the transformation function, `Optional::get`, extracts the value contained in the `Optional` returned. As we've said, here this is safe because the `reducing` collector will never return an `Optional.empty()`. The result is the following `Map`:

```
{FISH=salmon, OTHER=pizza, MEAT=pork}
```

It's quite common to use multiple nested collectors, and at first the way they interact may not always be obvious. Figure 6.6 helps you visualize how they work together. From the outermost layer and moving inward, note the following:

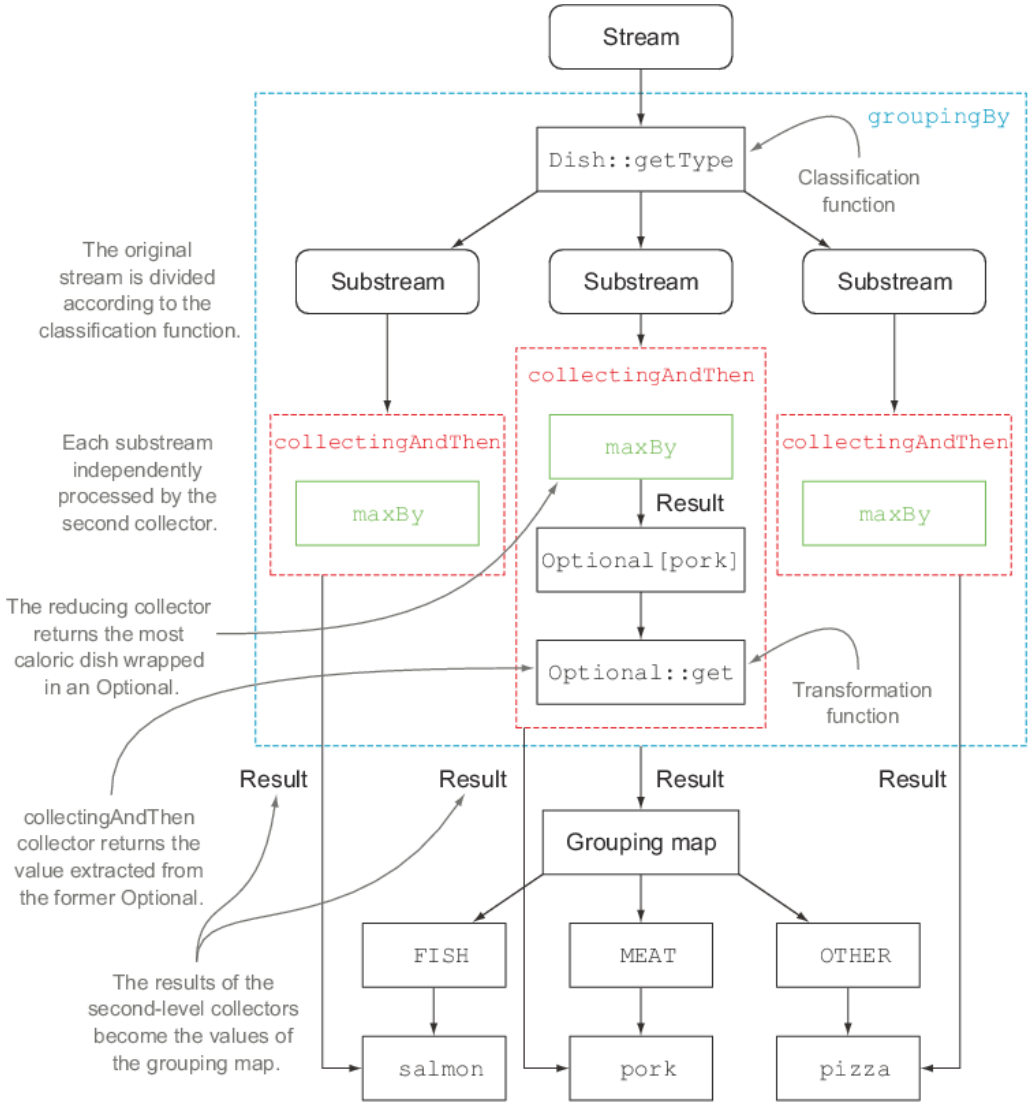


Figure 6.6 Combining the effect of multiple collectors by nesting one inside the other

- The collectors are represented by the dashed lines, so `groupingBy` is the outermost one and groups the menu stream into three substreams according to the different dishes' types.
- The `groupingBy` collector wraps the `collectingAndThen` collector, so each substream resulting from the grouping operation is further reduced by this second collector.

- The `collectingAndThen` collector wraps in turn a third collector, the `maxBy` one.
- The reduction operation on the substreams is then performed by the reducing collector, but the `collectingAndThen` collector containing it applies the `Optional::get` transformation function to its result.
- The three transformed values, being the highest-calorie `Dish`es for a given type (resulting from the execution of this process on each of the three substreams), will be the values associated with the respective classification keys, the types of `Dish`es, in the `Map` returned by the `groupingBy` collector.

OTHER EXAMPLES OF COLLECTORS USED IN CONJUNCTION WITH GROUPINGBY

More generally, the collector passed as second argument to the `groupingBy` factory method will be used to perform a further reduction operation on all the elements in the stream classified into the same group. For example, you could also reuse the collector created to sum the calories of all the dishes in the menu to obtain a similar result, but this time for each group of `Dish`es:

```
Map<Dish.Type, Integer> totalCaloriesByType =
                menu.stream().collect(groupingBy(Dish::getType,
                         summingInt(Dish::getCalories)));
```

Yet another collector, commonly used in conjunction with `groupingBy`, is one generated by the `mapping` method. This method takes two arguments: a function transforming the elements in a stream and a further collector accumulating the objects resulting from this transformation. Its purpose is to adapt a collector accepting elements of a given type to one working on objects of a different type, by applying a mapping function to each input element before accumulating them. To see a practical example of using this collector, suppose you want to know which `CaloricLevel`s are available in the menu for each type of Dish. You could achieve this result combining a `groupingBy` and a `mapping` collector as follows:

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
menu.stream().collect(
   groupingBy(Dish::getType, mapping(
    dish -> { if (dish.getCalories() <= 400) return CaloricLevel.DIET;
              else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
            else return CaloricLevel.FAT; },
    toSet() )));
```

Here the transformation function passed to the mapping method maps a `Dish` into its `CaloricLevel`, as you've seen before. The resulting stream of `CaloricLevel`s is then passed to a `toSet` collector, analogous to the `toList` one, but accumulating the elements of a stream into a `Set` instead of into a `List`, to keep only the distinct values. As in earlier examples, this mapping collector will then be used to collect the elements in each substream generated by the grouping function, allowing you to obtain as a result the following `Map`:

```
{OTHER=[DIET, NORMAL], MEAT=[DIET, NORMAL, FAT], FISH=[DIET, NORMAL]}
```

From this you can easily figure out your choices. If you're in the mood for fish and you're on a diet, you could easily find a dish; likewise, if you're very hungry and want something with lots of calories, you could satisfy your robust appetite by choosing something from the meat section of the menu. Note that in the previous example, there are no guarantees about what type of `Set` is returned. But by using `toCollection`, you can have more control. For example, you can ask for a `HashSet` by passing a constructor reference to it:

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
menu.stream().collect(
   groupingBy(Dish::getType, mapping(
    dish -> { if (dish.getCalories() <= 400) return CaloricLevel.DIET;
              else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
            else return CaloricLevel.FAT; },
    toCollection(HashSet::new) )));
```

6.4 Partitioning

Partitioning is a special case of grouping: having a predicate (a function returning a boolean), called a *partitioning function*, as a classification function. The fact that the partitioning function returns a boolean means the resulting grouping `Map` will have a `Boolean` as a key type and therefore there can be at most two different groups—one for `true` and one for `false`. For instance, if you're vegetarian or have invited a vegetarian friend to have dinner with you, you may be interested in partitioning the menu into vegetarian and nonvegetarian dishes:

```
Map<Boolean, List<Dish>> partitionedMenu =
             menu.stream().collect(partitioningBy(Dish::isVegetarian)); ❶
```

❶ **Partitioning function**

This will return the following `Map`:

```
{false=[pork, beef, chicken, prawns, salmon],
 true=[french fries, rice, season fruit, pizza]}
```

So you could retrieve all the vegetarian dishes by getting from this `Map` the value indexed with the key `true`:

```
List<Dish> vegetarianDishes = partitionedMenu.get(true);
```

Note that you could achieve the same result by just filtering the stream created from the menu `List` with the same predicate used for partitioning and then collecting the result in an additional `List`:

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

6.4.1 Advantages of partitioning

Partitioning has the advantage of keeping both lists of the stream elements, for which the application of the partitioning function returns `true` or `false`. So in the previous example, you

can obtain the `List` of the nonvegetarian `Dish`es by accessing the value of the key `false` in the `partitionedMenu Map`, using two separate filtering operations: one with the predicate and one with its negation. Also, as you already saw for grouping, the `partitioningBy` factory method has an overloaded version to which you can pass a second collector, as shown here:

```
Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =
menu.stream().collect(
        partitioningBy(Dish::isVegetarian,                          1
                       groupingBy(Dish::getType)));                 2
```

1 **Partitioning function**
2 **Second collector**

This will produce a two-level `Map`:

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

Here the grouping of the dishes by their type is applied individually to both of the substreams of vegetarian and nonvegetarian dishes resulting from the partitioning, producing a two-level `Map` that's similar to the one you obtained when you performed the two-level grouping in section 6.3.1. As another example, you can reuse your earlier code to find the most caloric dish among both vegetarian and nonvegetarian dishes:

```
Map<Boolean, Dish> mostCaloricPartitionedByVegetarian =
menu.stream().collect(
    partitioningBy(Dish::isVegetarian,
         collectingAndThen(maxBy(comparingInt(Dish::getCalories)),
                           Optional::get)));
```

That will produce the following result:

```
{false=pork, true=pizza}
```

We started this section by saying that you can think of partitioning as a special case of grouping. It's worth also noting that the `Map` implementation returned by `partitioningBy` is more compact and efficient as it only need to contain two keys: true and false. In fact, the internal implementation is a specialized `Map` with two fields. The analogies between the `groupingBy` and `partitioningBy` collectors don't end here; as you'll see in the next quiz, you can also perform multilevel partitioning in a way similar to what you did for grouping in section 6.3.1.

Quiz 6.2: Using partitioningBy

As you've seen, like the `groupingBy` collector, the `partitioningBy` collector can be used in combination with other collectors. In particular it could be used with a second `partitioningBy` collector to achieve a multilevel partitioning. What will be the result of the following multilevel partitionings?

```
1) menu.stream().collect(partitioningBy(Dish::isVegetarian,
```

```
                        partitioningBy(d -> d.getCalories() > 500)));
2) menu.stream().collect(partitioningBy(Dish::isVegetarian,
                        partitioningBy(Dish::getType)));
3) menu.stream().collect(partitioningBy(Dish::isVegetarian,
                        counting()));
```

Answer:

1) **This is a valid multilevel partitioning, producing the following two-level `Map`:**

```
{ false={false=[chicken, prawns, salmon], true=[pork, beef]},
  true={false=[rice, season fruit], true=[french fries, pizza]}}
```

2) **This won't compile because `partitioningBy` requires a predicate, a function returning a boolean. And the method reference `Dish::getType` can't be used as a predicate.**

3) **This counts the number of items in each partition, resulting in the following `Map`:**

```
{false=5, true=4}
```

To give one last example of how you can use the `partitioningBy` collector, we'll put aside the menu data model and look at something a bit more complex but also more interesting: partitioning numbers into prime and nonprime.

6.4.2 Partitioning numbers into prime and nonprime

Suppose you want to write a method accepting as argument an `int` *n* and partitioning the first *n* natural numbers into prime and nonprime. But first, it will be useful to develop a predicate that tests to see if a given candidate number is prime or not:

```
public boolean isPrime(int candidate) {
    return IntStream.range(2, candidate)                    (1)
                    .noneMatch(i -> candidate % i == 0);    (2)
}
```

(1) **Generate a range of natural numbers starting from and including 2 up to but excluding candidate.**
(2) **Return true if the candidate isn't divisible for any of the numbers in the stream.**

A simple optimization is to test only for factors less than or equal to the square root of the candidate:

```
public boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

Now the biggest part of the job is done. To partition the first *n* numbers into prime and nonprime, it's enough to create a stream containing those *n* numbers and reduce it with a `partitioningBy` collector using as predicate the `isPrime` method you just developed:

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
                    .collect(
                      partitioningBy(candidate -> isPrime(candidate)));
}
```

We've now covered all the collectors that can be created using the static factory methods of the `Collectors` class, showing practical examples of how they work. Table 6.1 brings them all together, with the type they return when applied to a `Stream<T>` and a practical example of their use on a `Stream<Dish>` named `menuStream`.

Table 6.1 The static factory methods of the `Collectors` class

Factory method	Returned type	Used to
`toList`	`List<T>`	Gather all the stream's items in a `List`.
Example use: `List<Dish> dishes = menuStream.collect(toList());`		
`toSet`	`Set<T>`	Gather all the stream's items in a `Set`, eliminating duplicates.
Example use: `Set<Dish> dishes = menuStream.collect(toSet());`		
`toCollection`	`Collection<T>`	Gather all the stream's items in the collection created by the provided supplier.
Example use: `Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new);`		
`counting`	`Long`	Count the number of items in the stream.
Example use: `long howManyDishes = menuStream.collect(counting());`		
`summingInt`	`Integer`	Sum the values of an Integer property of the items in the stream.
Example use: `int totalCalories = menuStream.collect(summingInt(Dish::getCalories));`		
`averagingInt`	`Double`	Calculate the average value of an `Integer` property of the items in the stream.
Example use: `double avgCalories = menuStream.collect(averagingInt(Dish::getCalories));`		
`summarizingInt`	`IntSummary-Statistics`	Collect statistics regarding an `Integer` property of the items in the stream, such as the maximum, minimum, total, and average.
Example use: `IntSummaryStatistics menuStatistics = menuStream.collect(summarizingInt(Dish::getCalories));`		
`joining`	`String`	Concatenate the strings resulting from the invocation of the `toString` method on each item of the stream.

Example use: `String shortMenu = menuStream.map(Dish::getName).collect(joining(", "));`

`maxBy`	`Optional<T>`	An `Optional` wrapping the maximal element in this stream according to the given comparator or `Optional.empty()` if the stream is empty.

Example use: `Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories)));`

`minBy`	`Optional<T>`	An `Optional` wrapping the minimal element in this stream according to the given comparator or `Optional.empty()` if the stream is empty.

Example use: `Optional<Dish> lightest = menuStream.collect(minBy(comparingInt(Dish::getCalories)));`

`reducing`	The type produced by the reduction operation	Reduce the stream to a single value starting from an initial value used as accumulator and iteratively combining it with each item of the stream using a `BinaryOperator`.

Example use: `int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum));`

`collectingAndThen`	The type returned by the transforming function	Wrap another collector and apply a transformation function to its result.

Example use: `int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size));`

`groupingBy`	`Map<K, List<T>>`	Group the items in the stream based on the value of one of their properties and use those values as keys in the resulting `Map`.

Example use: `Map<Dish.Type, List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType));`

`partitioningBy`	`Map<Boolean, List<T>>`	Partition the items in the stream based on the result of the application of a predicate to each of them.

Example use: `Map<Boolean, List<Dish>> vegetarianDishes = menuStream.collect(partitioningBy(Dish::isVegetarian));`

As we mentioned at the beginning of the chapter, all these collectors implement the `Collector` interface, so in the remaining part of the chapter we investigate this interface in

more detail. We investigate the methods in that interface and then explore how you can implement your own collectors.

6.5 The Collector interface

The `Collector` interface consists of a set of methods that provide a blueprint for how to implement specific reduction operations (that is, collectors). You've seen many collectors that implement the `Collector` interface, such as `toList` or `grouping-By`. This also implies that you're free to create customized reduction operations by providing your own implementation of the `Collector` interface. In section 6.6 we show how you can implement the `Collector` interface to create a collector to partition a stream of numbers into prime and nonprime more efficiently than what you've seen so far.

To get started with the `Collector` interface, we focus on one of the first collectors you encountered at the beginning of this chapter: the `toList` factory method, which gathers all the elements of a stream in a `List`. We said that you'll frequently use this collector in your day-to-day job, but it's also one that, at least conceptually, is straightforward to develop. Investigating in more detail how this collector is implemented is a good way to understand how the `Collector` interface is defined and how the functions returned by its methods are internally used by the `collect` method.

Let's start by taking a look at the definition of the `Collector` interface in the next listing, which shows the interface signature together with the five methods it declares.

Listing 6.4The `Collector` interface

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

In this listing, the following definitions apply:

- `T` is the generic type of the items in the stream to be collected.
- `A` is the type of the accumulator, the object on which the partial result will be accumulated during the collection process.
- `R` is the type of the object (typically, but not always, the collection) resulting from the collect operation.

For instance, you could implement a `ToListCollector<T>` class that gathers all the elements of a `Stream<T>` into a `List<T>` having the following signature

```
public class ToListCollector<T> implements Collector<T, List<T>, List<T>>
```

where, as we'll clarify shortly, the object used for the accumulation process will also be the final result of the collection process.

6.5.1 Making sense of the methods declared by Collector interface

We can now analyze one by one the five methods declared by the `Collector` interface. When we do so, you'll notice that each of the first four methods returns a function that will be invoked by the `collect` method, whereas the fifth one, `characteristics`, provides a set of characteristics that's a list of hints used by the `collect` method itself to know which optimizations (for example, parallelization) it's allowed to employ while performing the reduction operation.

MAKING A NEW RESULT CONTAINER: THE SUPPLIER METHOD

The `supplier` method has to return a `Supplier` of an empty accumulator—a parameterless function that when invoked creates an instance of an empty accumulator used during the collection process. Clearly, for a collector returning the accumulator itself as result, like our `ToListCollector`, this empty accumulator will also represent the result of the collection process when performed on an empty stream. In our `ToListCollector` the `supplier` will then return an empty `List` as follows:

```
public Supplier<List<T>> supplier() {
    return () -> new ArrayList<T>();
}
```

Note that you could also just pass a constructor reference:

```
public Supplier<List<T>> supplier() {
    return ArrayList::new;
}
```

ADDING AN ELEMENT TO A RESULT CONTAINER: THE ACCUMULATOR METHOD

The `accumulator` method returns the function that performs the reduction operation. When traversing the *n*th element in the stream, this function is applied with two arguments, the accumulator being the result of the reduction (after having collected the first *n*–1 items of the stream) and the *n*th element itself. The function returns `void` because the accumulator is modified in place, meaning that its internal state is changed by the function application to reflect the effect of the traversed element. For `ToListCollector`, this function merely has to add the current item to the list containing the already traversed ones:

```
public BiConsumer<List<T>, T> accumulator() {
    return (list, item) -> list.add(item);
```

You could instead use a method reference, which is more concise:

```
public BiConsumer<List<T>, T> accumulator() {
    return List::add;
}
```

APPLYING THE FINAL TRANSFORMATION TO THE RESULT CONTAINER: THE FINISHER METHOD

The `finisher` method has to return a function that's invoked at the end of the accumulation process, after having completely traversed the stream, in order to transform the accumulator object into the final result of the whole collection operation. Often, as in the case of the `ToListCollector`, the accumulator object already coincides with the final expected result. As a consequence, there's no need to perform a transformation, so the `finisher` method just has to return the `identity` function:

```
public Function<List<T>, List<T>> finisher() {
    return Function.identity();
}
```

These first three methods are enough to execute a sequential reduction of the stream that, at least from a logical point of view, could proceed as in figure 6.7. The implementation details are a bit more difficult in practice due to both the lazy nature of the stream, which could require a pipeline of other intermediate operations to execute before the `collect` operation, and the possibility, in theory, of performing the reduction in parallel.

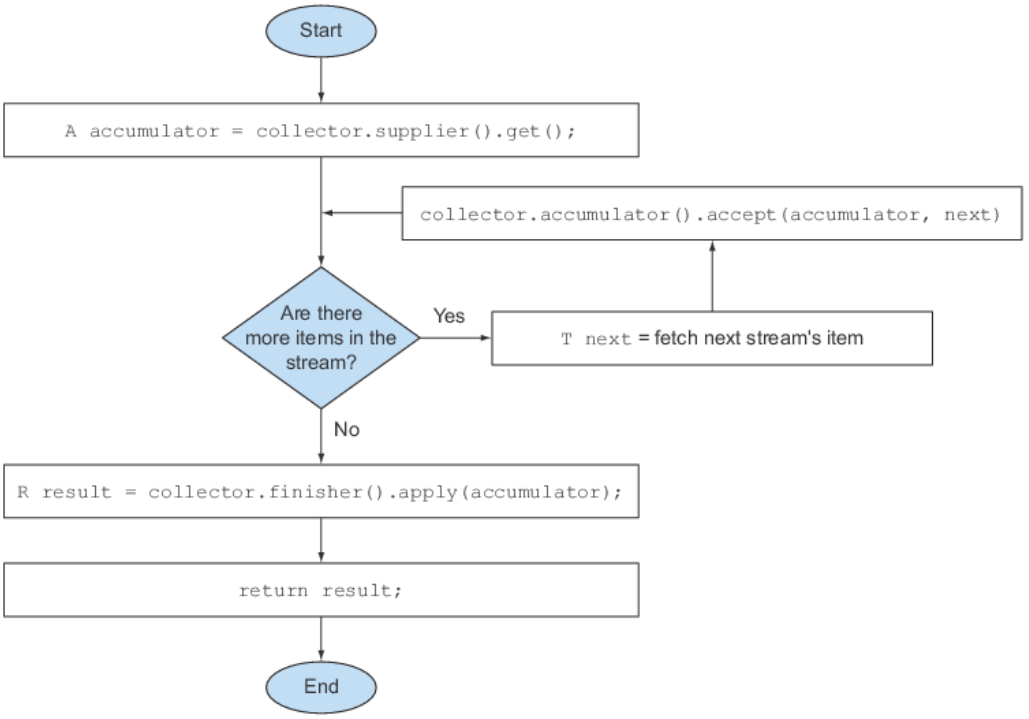


Figure 6.7 Logical steps of the sequential reduction process

MERGING TWO RESULT CONTAINERS: THE COMBINER METHOD

The `combiner` method, the last of the four methods that return a function used by the reduction operation, defines how the accumulators resulting from the reduction of different subparts of the stream are combined when the subparts are processed in parallel. In the `toList` case, the implementation of this method is simple; just add the list containing the items gathered from the second subpart of the stream to the end of the list obtained when traversing the first subpart:

```
public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
        list1.addAll(list2);
        return list1; }
}
```

The addition of this fourth method allows a parallel reduction of the stream. This uses the fork/join framework introduced in Java 7 and the `Spliterator` abstraction that you'll learn about in the next chapter. It follows a process similar to the one shown in figure 6.8 and described in detail here:

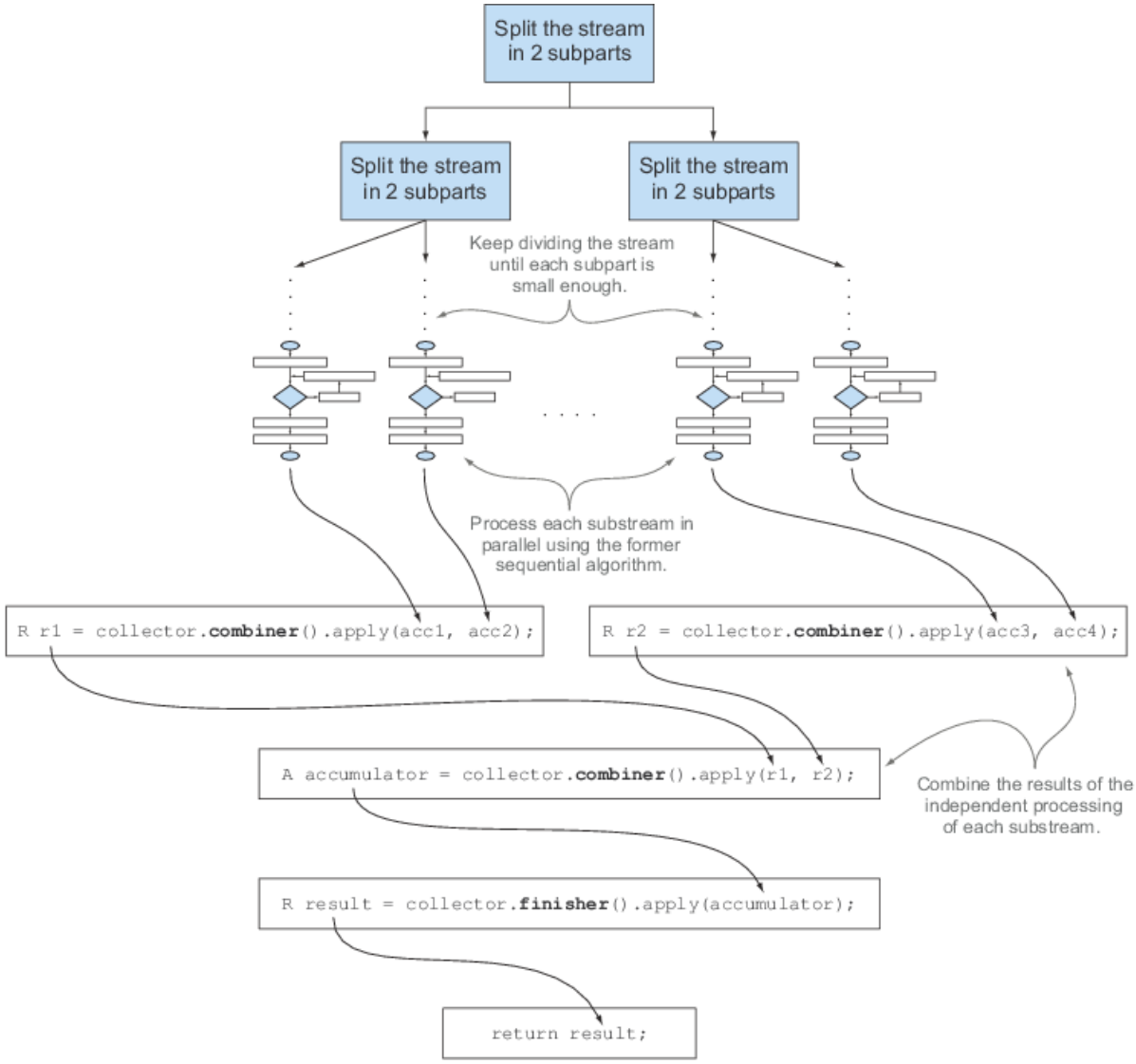


Figure 6.8 Parallelizing the reduction process using the `combiner` method

- The original *stream* is recursively split in substreams until a condition defining whether a stream needs to be further divided becomes false (parallel computing is often slower than sequential computing when the units of work being distributed are too small, and it's pointless to generate many more parallel tasks than you have processing cores).
- At this point all *substreams* can be processed in parallel, each of them using the

sequential reduction algorithm shown in figure 6.7.

- Finally, all the partial results are combined pairwise using the function returned by the `combiner` method of the collector. This is done by combining results corresponding to substreams associated with each split of the original stream.

THE CHARACTERISTICS METHOD

The last method, `characteristics`, returns an immutable set of `Characteristics`, defining the behavior of the collector—in particular providing hints about whether the stream can be reduced in parallel and which optimizations are valid when doing so. `Characteristics` is an enumeration containing three items:

- `UNORDERED`—The result of the reduction isn't affected by the order in which the items in the stream are traversed and accumulated.
- `CONCURRENT`—The `accumulator` function can be called concurrently from multiple threads, and then this collector can perform a parallel reduction of the stream. If the collector isn't also flagged as `UNORDERED`, it can perform a parallel reduction only when it's applied to an unordered data source.
- `IDENTITY_FINISH`—This indicates the function returned by the finisher method is the identity one, and its application can be omitted. In this case, the accumulator object is directly used as the final result of the reduction process. This also implies that it's safe to do an unchecked cast from the accumulator `A` to the result `R`.

The `ToListCollector` developed so far is `IDENTITY_FINISH`, because the `List` used to accumulate the elements in the stream is already the expected final result and doesn't need any further transformation, but it isn't `UNORDERED` because if you apply it to an ordered stream you want this ordering to be preserved in the resulting `List`. Finally, it's `CONCURRENT`, but following what we just said, the stream will be processed in parallel only if its underlying data source is unordered.

6.5.2 Putting them all together

The five methods analyzed in the preceding subsection are everything you need to develop your own `ToListCollector`, so you can implement it by putting all of them together, as the next listing shows.

Listing 6.5 The `ToListCollector`

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collector;
import static java.util.stream.Collector.Characteristics.*;
public class ToListCollector<T> implements Collector<T, List<T>, List<T>> {
    @Override
    public Supplier<List<T>> supplier() {
        return ArrayList::new;                                    (1)
    }
```

```
    @Override
    public BiConsumer<List<T>, T> accumulator() {
        return List::add;                                          ❷
    }
    @Override
    public Function<List<T>, List<T>> finisher() {
        return Function.indentity();                               ❸
    }
    @Override
    public BinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);                                   ❹
            return list1;                                          ❺
        };
    }
    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(
            IDENTITY_FINISH, CONCURRENT));                         ❻
    }
}
```

❶ Creates the collection operation starting point
❷ Accumulates the traversed item, modifying the accumulator in place
❸ Identity function
❹ Modifies the first accumulator, combining it with the content of the second one
❺ Returns the modified first accumulator
❻ Flags the collector as IDENTITY_FINISH and CONCURRENT

Note that this implementation isn't identical to the one returned by the `Collectors .toList` method, but it differs only in some minor optimizations. These optimizations are mostly related to the fact that the collector provided by the Java API uses the `Collections.emptyList()` singleton when it has to return an empty list. This means that it could be safely used in place of the original Java as an example to gather a list of all the `Dish`es of a menu stream:

```
List<Dish> dishes = menuStream.collect(new ToListCollector<Dish>());
```

The remaining difference from this and the standard

```
List<Dish> dishes = menuStream.collect(toList());
```

formulation is that `toList` is a factory, whereas you have to use `new` to instantiate your `ToListCollector`.

PERFORMING A CUSTOM COLLECT WITHOUT CREATING A COLLECTOR IMPLEMENTATION

In the case of an `IDENTITY_FINISH` collection operation, there's a further possibility of obtaining the same result without developing a completely new implementation of the `Collector` interface. `Stream` has an overloaded `collect` method accepting the three other functions—`supplier`, `accumulator`, and `combiner`—having exactly the same semantics as the

ones returned by the corresponding methods of the `Collector` interface. So, for instance, it's possible to collect in a `List` all the items in a stream of dishes as follows:

```
List<Dish> dishes = menuStream.collect(
                        ArrayList::new,          1
                        List::add,               2
                        List::addAll);           3
```

1. Supplier
2. Accumulator
3. Combiner

We believe that this second form, even if more compact and concise than the former one, is rather less readable. Also, developing an implementation of your custom collector in a proper class promotes its reuse and helps avoid code duplication. It's also worth noting that you're not allowed to pass any `Characteristics` to this second `collect` method, so it always behaves as an `IDENTITY_FINISH` and `CONCURRENT` but not `UNORDERED` collector.

In the next section, you'll take your new knowledge of implementing collectors to the next level. You'll develop your own custom collector for a more complex but hopefully more specific and compelling use case.

6.6 Developing your own collector for better performance

In section 6.4, where we discussed partitioning, you created a collector, using one of the many convenient factory methods provided by the `Collectors` class, which divides the first *n* natural numbers into primes and nonprimes, as shown in the following listing.

Listing 6.6 Partitioning the first *n* natural numbers into primes and nonprimes

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
return IntStream.rangeClosed(2, n).boxed()
                .collect(partitioningBy(candidate -> isPrime(candidate));
}
```

There you achieved an improvement over the original `isPrime` method by limiting the number of divisors to be tested against the candidate prime to those not bigger than the candidate's square root:

```
public boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

Is there a way to obtain even better performances? The answer is yes, but for this you'll have to develop a custom collector.

6.6.1 Divide only by prime numbers

One possible optimization is to test only if the candidate number is divisible by prime numbers. It's pointless to test it against a divisor that's not itself prime! So you can limit the test to only the prime numbers found before the current candidate. The problem with the predefined collectors you've used so far, and the reason you have to develop a custom one, is that during the collecting process you don't have access to the partial result. This means that when testing whether a given candidate number is prime or not, you don't have access to the list of the other prime numbers found so far.

Suppose you had this list; you could pass it to the `isPrime` method and rewrite it as follows:

```
public static boolean isPrime(List<Integer> primes, int candidate) {
    return primes.stream().noneMatch(i -> candidate % i == 0);
}
```

Also, you should implement the same optimization you used before and test only with primes smaller than the square root of the candidate number. So you need a way to stop testing whether the candidate is divisible by a prime as soon as the next prime is greater than the candidate's root. You can easily do this by using the Stream's `takeWhile` method.

```
public static boolean isPrime(List<Integer> primes, int candidate){
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return primes.stream()
                 .takeWhile(i -> i <= candidateRoot)
                 .noneMatch(i -> candidate % i == 0);
}
```

Quiz 6.3: Simulating takeWhile in Java 8

The `takeWhile` method was introduced in Java 9, so unfortunately you cannot use this solution if you are still using Java 8. How could you work round this limitation and achieve something similar also in Java 8?

Answer:

You could implement your own `takeWhile` method which, given a sorted list and a predicate, returns the longest prefix of this list whose elements satisfy the predicate:

```
public static <A> List<A> takeWhile(List<A> list, Predicate<A> p) {
    int i = 0;
    for (A item : list) {
            if (!p.test(item)) {                          (1)
                    return list.subList(0, i);            (2)
            }
            i++;
    }
    return list;                                          (3)
}
```

(1) **Check if the current item in the list satisfies the Predicate.**
(2) **If it doesn't, return the sublist prefix until the item before the tested one.**
(3) **All the items in the list satisfy the Predicate, so return the list itself.**

Using this method, you can rewrite the isPrime method and once again testing only the candidate prime against only the primes that are not greater than its square root:

```
public static boolean isPrime(List<Integer> primes, int candidate){
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return takeWhile(primes, i -> i <= candidateRoot)
                 .stream()
                 .noneMatch(p -> candidate % p == 0);
}
```

Note that, unlike the one provided by the Stream API, this implementation of `takeWhile` is eager. When possible always prefer the Java 9 Stream's lazy version of `takeWhile` so it can be merged with the noneMatch operation.

With this new `isPrime` method in hand, you're now ready to implement your own custom collector. First, you need to declare a new class that implements the `Collector` interface. Then, you need to develop the five methods required by the `Collector` interface.

STEP 1: DEFINING THE COLLECTOR CLASS SIGNATURE

Let's start with the class signature, remembering that the `Collector` interface is defined as

```
public interface Collector<T, A, R>
```

where `T`, `A`, and `R` are respectively the type of the elements in the stream, the type of the object used to accumulate partial results, and the type of the final result of the `collect` operation. In this case, you want to collect streams of `Integer`s while both the accumulator and the result types are `Map<Boolean, List<Integer>>` (the same `Map` you obtained as a result of the former partitioning operation in listing 6.6), having as keys `true` and `false` and as values respectively the `List`s of prime and nonprime numbers:

```
public class PrimeNumbersCollector
             implements Collector<Integer,                          (1)
                                  Map<Boolean, List<Integer>>,      (2)
                                  Map<Boolean, List<Integer>>>      (3)
```

(1) **The type of the elements in the stream**
(2) **The type of the accumulator**
(3) **The type of the result of the collect operation**

STEP 2: IMPLEMENTING THE REDUCTION PROCESS

Next, you need to implement the five methods declared in the `Collector` interface. The `supplier` method has to return a function that when invoked creates the accumulator:

```
public Supplier<Map<Boolean, List<Integer>>> supplier() {
    return () -> new HashMap<Boolean, List<Integer>>() {{
        put(true, new ArrayList<Integer>());
        put(false, new ArrayList<Integer>());
    }};
}
```

Here you're not only creating the `Map` that you'll use as the accumulator, but you're also initializing it with two empty lists under the `true` and `false` keys. This is where you'll add respectively the prime and nonprime numbers during the collection process. The most important method of your collector is the `accumulator` method, because it contains the logic defining how the elements of the stream have to be collected. In this case, it's also the key to implementing the optimization we described previously. At any given iteration you can now access the partial result of the collection process, which is the accumulator containing the prime numbers found so far:

```
public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
    return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
            acc.get( isPrime(acc.get(true), candidate) )              (1)
               .add(candidate);                                       (2)
    };
}
```

(1) **Get the list of prime or nonprime numbers depending on the result of isPrime.**
(2) **Add the candidate to the appropriate list.**

In this method, you invoke the `isPrime` method, passing to it (together with the number for which you want to test whether it's prime or not) the list of the prime numbers found so far (these are the values indexed by the `true` key in the accumulating `Map)`. The result of this invocation is then used as key to get the list of either the prime or nonprime numbers so you can add the new candidate to the right list.

STEP 3: MAKING THE COLLECTOR WORK IN PARALLEL (IF POSSIBLE)

The next method has to combine two partial accumulators in the case of a parallel collection process, so in this case it just has to merge the two `Map`s by adding all the numbers in the prime and nonprime lists of the second `Map` to the corresponding lists in the first `Map`:

```
public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
    return (Map<Boolean, List<Integer>> map1,
            Map<Boolean, List<Integer>> map2) -> {
                map1.get(true).addAll(map2.get(true));
                map1.get(false).addAll(map2.get(false));
                return map1;
    };
}
```

Note that in reality this collector can't be used in parallel, because the algorithm is inherently sequential. This means the `combiner` method won't ever be invoked, and you could leave its implementation empty (or better, throw an `UnsupportedOperation-Exception`). We decided to implement it anyway only for completeness.

STEP 4: THE FINISHER METHOD AND THE COLLECTOR'S CHARACTERISTIC METHOD

The implementation of the last two methods is quite straightforward: as we said, the `accumulator` coincides with the collector's result so it won't need any further transformation, and the `finisher` method returns the `identity` function:

```
public Function<Map<Boolean, List<Integer>>,
                Map<Boolean, List<Integer>>> finisher() {
        return Function.identity();
}
```

As for the characteristic method, we already said that it's neither `CONCURRENT` nor `UNORDERED` but is `IDENTITY_FINISH`:

```
public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
}
```

The following listing shows the final implementation of `PrimeNumbersCollector`.

Listing 6.7 The `PrimeNumbersCollector`

```
public class PrimeNumbersCollector
    implements Collector<Integer,
           Map<Boolean, List<Integer>>,
           Map<Boolean, List<Integer>>> {
    @Override
    public Supplier<Map<Boolean, List<Integer>>> supplier() {
        return () -> new HashMap<Boolean, List<Integer>>() {{           (1)
             put(true, new ArrayList<Integer>());
             put(false, new ArrayList<Integer>());
          }};
    }
    @Override
    public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
        return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
            acc.get( isPrime( acc.get(true),                            (2)
               candidate) )
               .add(candidate);                                         (3)
        };
    }
    @Override
    public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
            return (Map<Boolean, List<Integer>> map1,
                    Map<Boolean, List<Integer>> map2) -> {              (4)
                            map1.get(true).addAll(map2.get(true));
                            map1.get(false).addAll(map2.get(false));
                            return map1;
                 };
    }
    @Override
    public Function<Map<Boolean, List<Integer>>,
                    Map<Boolean, List<Integer>>> finisher() {
        return Function.identity();                                     (5)
    }
    @Override
```

```
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH)); (6)
    }
}
```

(1) Start the collection process with a Map containing two empty Lists.
(2) Pass to the isPrime method the list of already found primes.
(3) Get from the Map the list of prime or nonprime numbers, according to what the isPrime method returned, and add to it the current candidate.
(4) Merge the second Map into the first one.
(5) No transformation is necessary at the end of the collection process, so terminate it with the identity function.
(6) This collector is IDENTITY_FINISH but neither UNORDERED nor CONCURRENT because it relies on the fact that prime numbers are discovered in sequence.

You can now use this new custom collector in place of the former one created with the `partitioningBy` factory method in section 6.4 and obtain exactly the same result:

```
public Map<Boolean, List<Integer>>
                        partitionPrimesWithCustomCollector(int n) {
        return IntStream.rangeClosed(2, n).boxed()
                        .collect(new PrimeNumbersCollector());
}
```

6.6.2 Comparing collectors' performances

The collector created with the `partitioningBy` factory method and the custom one you just developed are functionally identical, but did you achieve your goal of improving the performance of the `partitioningBy` collector with your custom one? Let's write a quick harness to check this:

```
public class CollectorHarness {
    public static void main(String[] args) {
        long fastest = Long.MAX_VALUE;
        for (int i = 0; i < 10; i++) {                                  (1)
            long start = System.nanoTime();
            partitionPrimes(1_000_000);                                 (2)
            long duration = (System.nanoTime() - start) / 1_000_000;    (3)
            if (duration < fastest) fastest = duration;                 (4)
        }
        System.out.println(
            "Fastest execution done in " + fastest + " msecs");
    }
}
```

(1) Run the test 10 times.
(2) Partition into primes and nonprimes the first million natural numbers.
(3) Take the duration in milliseconds.
(4) Check if this execution has been the fastest one.

Note that a more scientific benchmarking approach would be to use a framework such as JMH, but we didn't want to add the complexity of using such a framework here and, for this use case, the results provided by this small benchmarking class are accurate enough. This class partitions the first million natural numbers into primes and nonprimes, invoking the method

using the collector created with the `partitioningBy` factory method 10 times and registering the fastest execution. Running it on an Intel i5 2.4 GHz, it prints the following result:

```
Fastest execution done in 4716 msecs
```

Now replace `partitionPrimes` with `partitionPrimesWithCustomCollector` in the harness, in order to test the performances of the custom collector you developed. Now the program prints

```
Fastest execution done in 3201 msecs
```

Not bad! This means you didn't waste your time developing this custom collector for two reasons: first, you learned how to implement your own collector when you need it, and second, you achieved a performance improvement of around 32%.

Finally, it's important to note that, as you did for the `ToListCollector` in listing 6.5, it's possible to obtain the same result by passing the three functions implementing the core logic of `PrimeNumbersCollector` to the overloaded version of the `collect` method, taking them as arguments:

```
public Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector
          (int n) {
    IntStream.rangeClosed(2, n).boxed()
        .collect(
                () -> new HashMap<Boolean, List<Integer>>() {{ ❶
                   put(true, new ArrayList<Integer>());
                   put(false, new ArrayList<Integer>());
              }},
             (acc, candidate) -> {                                  ❷
                acc.get( isPrime(acc.get(true), candidate) )
                   .add(candidate);
            },
              (map1, map2) -> {                                     ❸
                map1.get(true).addAll(map2.get(true));
                map1.get(false).addAll(map2.get(false));
            });
}
```

❶ **Supplier**
❷ **Accumulator**
❸ **Combiner**

As you can see, in this way you can avoid creating a completely new class that implements the `Collector` interface; the resulting code is more compact, even if it's also probably less readable and certainly less reusable.

6.7 Summary

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

- `collect` is a terminal operation that takes as argument various recipes (called collectors) for accumulating the elements of a stream into a summary result.
- Predefined collectors include reducing and summarizing stream elements into a single

value, such as calculating the minimum, maximum, or average. Those collectors are summarized in table 6.1.

- Predefined collectors let you group elements of a stream with `groupingBy` and partition elements of a stream with `partitioningBy`.
- Collectors compose effectively to create multilevel groupings, partitions, and reductions.
- You can develop your own collectors by implementing the methods defined in the `Collector` interface.