Lambdas, streams, and functional-style programming

Java 8 IN ACTION

Raoul-Gabriel Urma
Mario Fusco
Alan Mycroft

SAMPLE CHAPTER

MANNING

Java 8 in Action

by Raoul-Gabriel Urma
Mario Fusco
Alan Mycroft

Chapter 4

brief contents

4 Introducing streams

This chapter covers

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

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

- Much business logic entails database-like operations such as *grouping* a list of dishes by category (for example, all vegetarian dishes) or *finding* the most expensive dish. How many times do you find yourself reimplementing these operations using iterators? Most databases let you specify such operations declaratively. For example, the following SQL query lets you select the names

of dishes that are low in calories: `SELECT name FROM dishes WHERE calorie < 400`. As you can see, you don't need to implement how to filter using the attributes of a dish (for example, using an iterator and an accumulator). Instead, you express only what you expect. This basic idea means that you worry less about how to explicitly implement such queries—it's handled for you! Why can't you do something similar with collections?

- How would you process a large collection of elements? To gain performance you'd need to process it in parallel and leverage multicore architectures. But writing parallel code is complicated in comparison to working with iterators. In addition, it's no fun to debug!

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

4.1 What are streams?

Streams are an update to the Java API that lets you manipulate collections of data in a declarative way (you express a query rather than code an ad hoc implementation for it). For now you can think of them as fancy iterators over a collection of data. In addition, streams can be processed in parallel *transparently*, without you having to write any multithreaded code! We explain in detail in chapter 7 how streams and parallelization work. Here's a taste of the benefits of using streams: compare the following code to return the names of dishes that are low in calories, sorted by number of calories, first in Java 7 and then in Java 8 using streams. Don't worry about the Java 8 code too much; we explain it in detail in the next sections!

Before (Java 7):

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

Filter the elements using an accumulator.

Sort the dishes with an anonymous class.

Process the sorted list to select the names of dishes.

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

After (Java 8):

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
```

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

Store all the names in a `List`.

Select dishes that are below 400 calories.

Sort them by calories.

Extract the names of these dishes.

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

```
List<String> lowCaloricDishesName =
                menu.parallelStream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dishes::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

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

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

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!

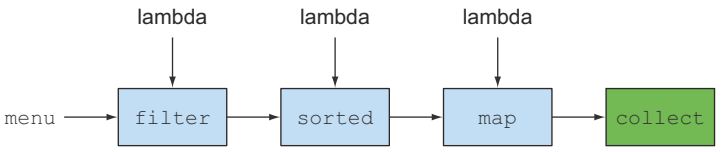


Figure 4.1 Chaining stream operations forming a stream pipeline

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

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

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

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

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

Other libraries: Guava, Apache, and lambdaj

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

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

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

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

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

```
List<Dish> menu = Arrays.asList(
    new Dish("pork", false, 800, Dish.Type.MEAT),
    new Dish("beef", false, 700, Dish.Type.MEAT),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
    new Dish("french fries", true, 530, Dish.Type.OTHER),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("season fruit", true, 120, Dish.Type.OTHER),
    new Dish("pizza", true, 550, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("salmon", false, 450, Dish.Type.FISH) );
```

where a Dish is an immutable class defined as

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

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

    public String getName() {
        return name;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public int getCalories() {
        return calories;
    }

    public Type getType() {
        return type;
    }

    @Override
    public String toString() {
        return name;
    }

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

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

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

4.2 Getting started with streams

We start our discussion of streams with collections, because that's the simplest way to begin working with streams. Collections in Java 8 support a new stream method that

returns a stream (the interface definition is available in `java.util.stream.Stream`). You'll later see that you can also get streams in various other ways (for example, generating stream elements from a numeric range or from I/O resources).

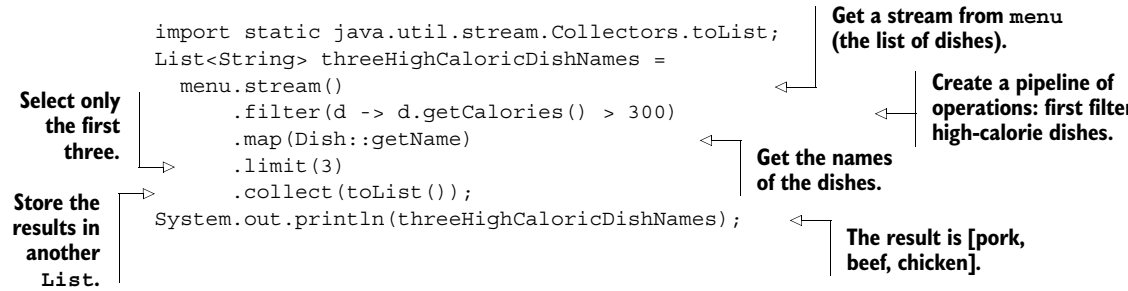
So first, what exactly is a *stream*? A short definition is "a sequence of elements from a source that supports data processing operations." Let's break down this definition step by step:

- *Sequence of elements*—Like a collection, a stream provides an interface to a sequenced set of values of a specific element type. Because collections are data structures, they're mostly about storing and accessing elements with specific time/space complexities (for example, an `ArrayList` vs. a `LinkedList`). But streams are about expressing computations such as `filter`, `sorted`, and `map` that you saw earlier. Collections are about data; streams are about computations. We explain this idea in greater detail in the coming sections.
- *Source*—Streams consume from a data-providing source such as collections, arrays, or I/O resources. Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.
- *Data processing operations*—Streams support database-like operations and common operations from functional programming languages to manipulate data, such as `filter`, `map`, `reduce`, `find`, `match`, `sort`, and so on. Stream operations can be executed either sequentially or in parallel.

In addition, stream operations have two important characteristics:

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

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



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*

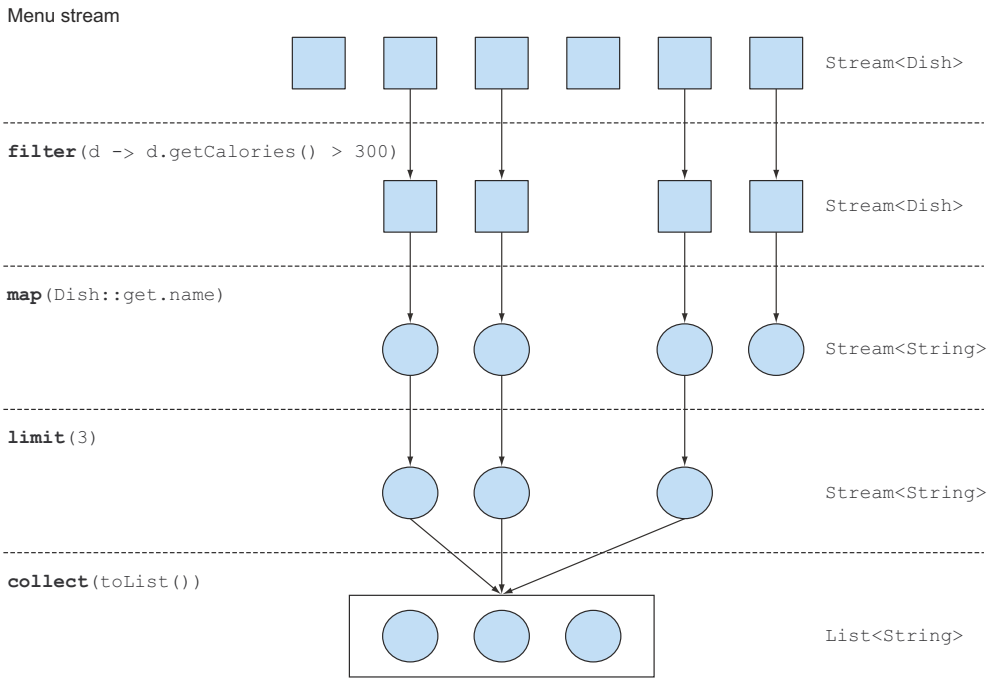


Figure 4.2 Filtering a menu using a stream to find out three high-calorie dish names

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:

- `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.

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.

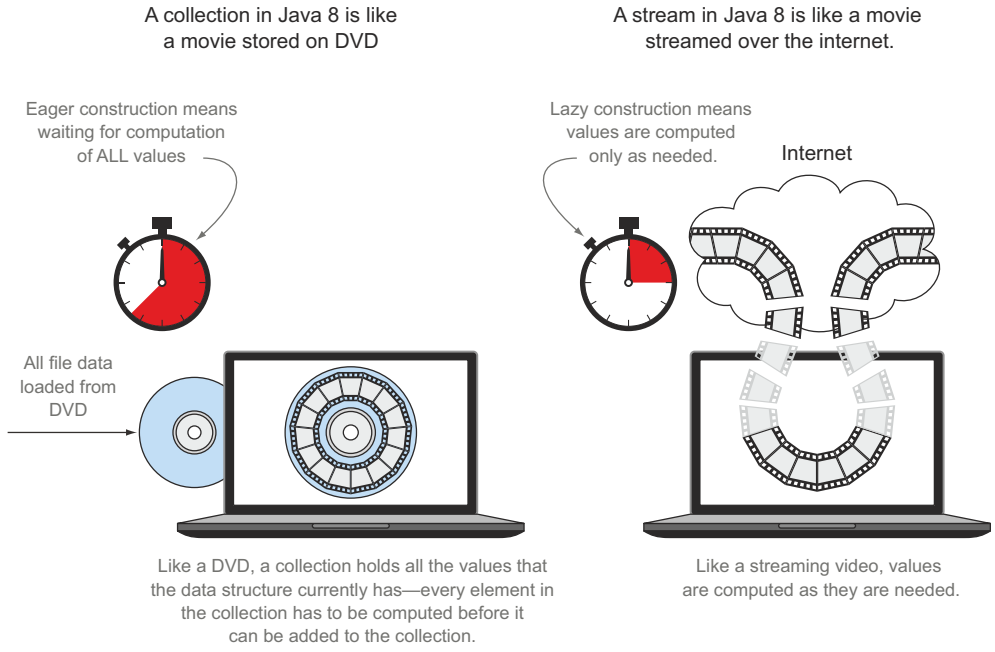


Figure 4.3 Streams vs. collections

4.3.1 Traversable only once

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

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

Prints each word in the title.

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

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

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

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

4.3.2 External vs. internal iteration

Using the `Collection` interface requires iteration to be done by the user (for example, using `for-each`); this is called *external iteration*. The Streams library by contrast uses *internal iteration*—it does the iteration for you and takes care of storing the resulting stream value somewhere; you merely provide a function saying what's to be done. The following code listings illustrate this difference.

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

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

Explicitly iterate the list of menu sequentially.

Extract the name and add it to an accumulator.

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

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

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

Iterating explicitly

Listing 4.3 Streams: internal iteration

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

Start executing the pipeline of operations; no iteration!

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

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

You: "Sofia, let's put the toys away. Is there a toy on the ground?"
Sofia: "Yes, the ball."
You: "Okay, put the ball in the box. Is there something else?"
Sofia: "Yes, there's my doll."
You: "Okay, put the doll in the box. Is there something else?"
Sofia: "Yes, there's my book."
You: "Okay, put the book in the box. Is there something else?"
Sofia: "No, nothing else."
You: "Fine, we're finished."

This is exactly what you do every day with your Java collections. You iterate a collection *externally*, explicitly pulling out and processing the items one by one. It would be far better if you could just tell Sofia, "Put all the toys that are on the floor inside the box." There are two other reasons why an internal iteration is preferable: first, Sofia could choose to take at the same time the doll with one hand and the ball with the other, and second, she could decide to take the objects closest to the box first and then the others. In the same way, using an internal iteration, the processing of items could be transparently done in parallel or in a different order that may be more optimized. These optimizations are difficult if you iterate the collection externally as you're used to doing in Java. This may seem like nit-picking, but it's much of the raison-d'être of Java 8's introduction of streams—the internal iteration in the Streams library can automatically choose a data representation and implementation of parallelism to match your hardware. By contrast, once you've chosen external iteration by writing `for-each`, then you've essentially committed to self-manage any parallelism. (*Self-managing* in practice means either "one fine day we'll parallelize this" or "starting the long and arduous battle involving tasks and `synchronized`".) Java 8 needed an interface like `Collection` but without iterators, ergo `Stream`! Figure 4.4 illustrates the difference between a stream (internal iteration) and a collection (external iteration).

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

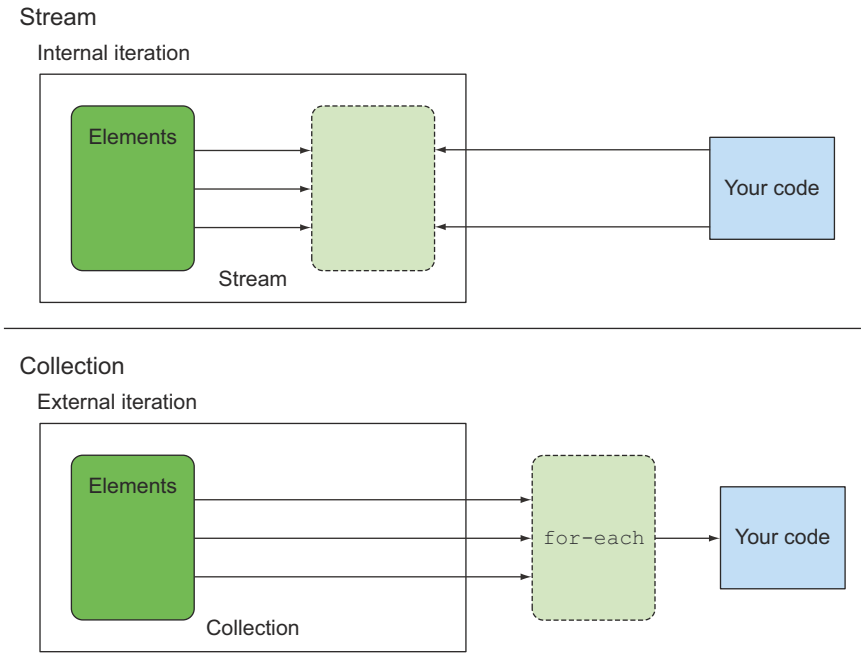


Figure 4.4 Internal vs. external iteration

list of operations you can use to express complicated data processing queries. We'll briefly look at this list of operations now and explore them in more detail with examples in the next chapter.

4.4 Stream operations

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

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

Get a stream from the list of dishes.

Intermediate operation.

Intermediate operation.

Intermediate operation.

Converts the `Stream` into a `List`.

You can see two groups of operations:

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

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

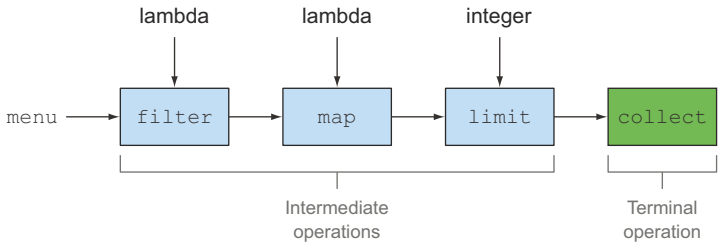


Figure 4.5 Intermediate vs. terminal operations

4.4.1 Intermediate operations

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

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

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

Printing the dishes as they're filtered

Printing the dishes as you extract their names

This code when executed will print the following:

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

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

4.4.2 Terminal operations

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

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

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

Quiz 4.1: Intermediate vs. terminal operations

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

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

Answer:

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

4.4.3 Working with streams

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

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

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

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!

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

Table 4.1 Intermediate operations

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

Table 4.2 Terminal operations

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

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

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

4.5 Summary

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

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

JAVA

Java 8 IN ACTION

Urma • Fusco • Mycroft

Every new version of Java is important, but Java 8 is a game changer. With Java 8's functional features you can write more concise code in less time, and also automatically benefit from multicore architectures. It's time to dig in!

Java 8 in Action is a clearly written guide to the new features of Java 8. It begins with a practical introduction to lambdas, using real-world Java code. Next, it covers the new Streams API and shows how you can use it to make collection-based code radically easier to understand and maintain. It also explains other major Java 8 features including default methods, `Optional`, `CompletableFuture`, and the new Date and Time API.

What's Inside

- How to use Java 8's powerful new features
- Writing effective multicore-ready applications
- Refactoring, testing, and debugging
- Adopting functional-style programming
- Quizzes and quick-check questions

This book is written for programmers familiar with Java and basic OO programming.

Raoul-Gabriel Urma is a software engineer, speaker, trainer, and PhD candidate at the University of Cambridge. **Mario Fusco** is an engineer at Red Hat and creator of the lambdaj library. **Alan Mycroft** is a professor at Cambridge and cofounder of the Raspberry Pi Foundation.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/Java8inAction

"A great and concise guide to what's new in Java 8, with plenty of examples to get you going in a hurry."
—Jason Lee, Oracle

"The best guide to Java 8 that will ever be written!"
—William Wheeler
ProData Computer Systems

"The new Streams API and lambda examples are especially useful."
—Steve Rogers, CGTek, Inc.

"A must-have to get functional with Java 8."
—Mayur S. Patil
MIT Academy of Engineering

ISBN 13: 978-1-617291-99-9
ISBN 10: 1-617291-99-4
54999
9 781617 291999

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]